

# Chapter 7

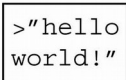


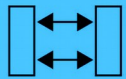
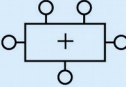
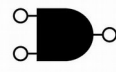
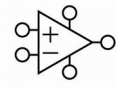


## ***Digital Design and Computer Architecture, 2<sup>nd</sup> Edition***

---

David Money Harris and Sarah L. Harris

# Chapter 7 :: Topics

- Introduction
- Performance Analysis
- Single-Cycle Processor
- Multicycle Processor
- Pipelined Processor
- Exceptions
- Advanced Microarchitecture

Application Software	
Operating Systems	
Architecture	
Micro-architecture	
Logic	
Digital Circuits	
Analog Circuits	
Devices	
Physics	

# Introduction

- **Microarchitecture:** how to implement an architecture in hardware
- **Processor:**
  - **Datapath:** functional blocks
  - **Control:** control signals

Application Software	programs
Operating Systems	device drivers
Architecture	instructions registers
Micro-architecture	datapaths controllers
Logic	adders memories
Digital Circuits	AND gates NOT gates
Analog Circuits	amplifiers filters
Devices	transistors diodes
Physics	electrons



# Microarchitecture

- Multiple implementations for a single architecture:
  - **Single-cycle:** Each instruction executes in a single cycle
  - **Multicycle:** Each instruction is broken into series of shorter steps
  - **Pipelined:** Each instruction broken up into series of steps & multiple instructions execute at once

# Processor Performance

- Program execution time

Execution Time = (#instructions)(cycles/instruction)(seconds/cycle)

- Definitions:
  - CPI: Cycles/instruction
  - clock period: seconds/cycle
  - IPC: instructions/cycle = IPC
- Challenge is to satisfy constraints of:
  - Cost
  - Power
  - Performance

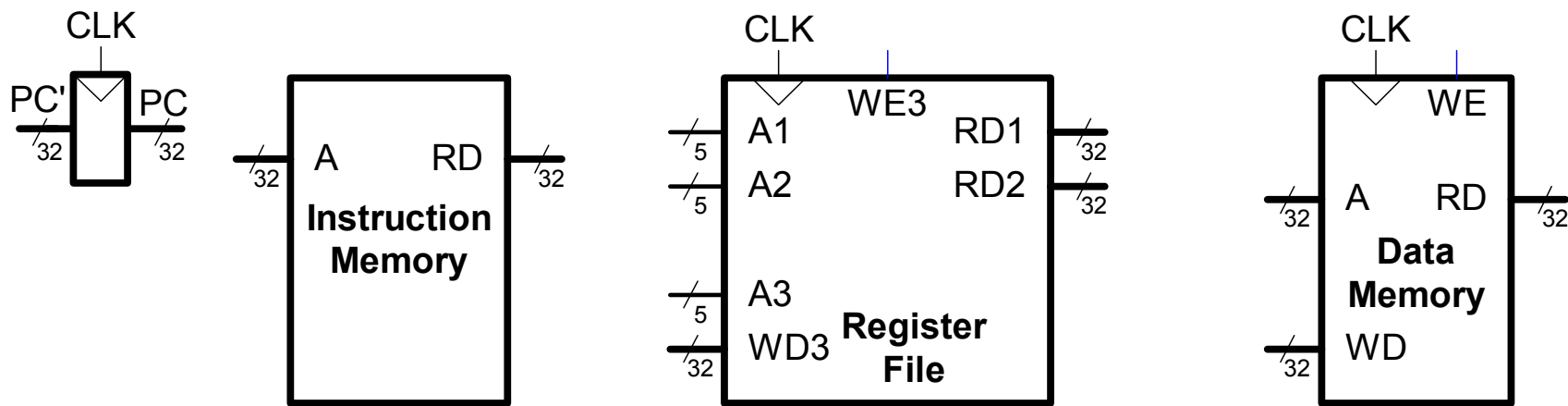
# MIPS Processor

- Consider subset of MIPS instructions:
  - R-type instructions: `and`, `or`, `add`, `sub`, `sllt`
  - Memory instructions: `lw`, `sw`
  - Branch instructions: `beq`

# Architectural State

- Determines everything about a processor:
  - PC
  - 32 registers
  - Memory

# MIPS State Elements



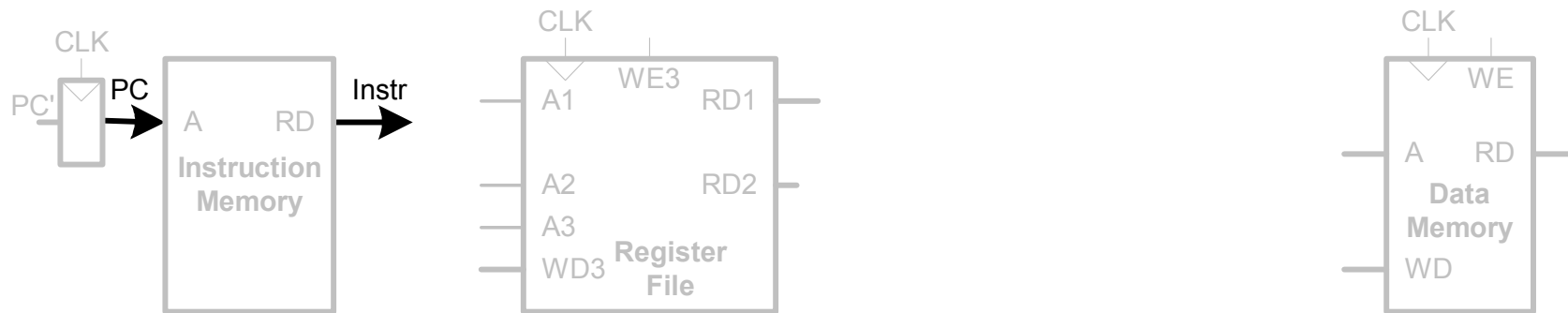


# Single-Cycle MIPS Processor

- Datapath
- Control

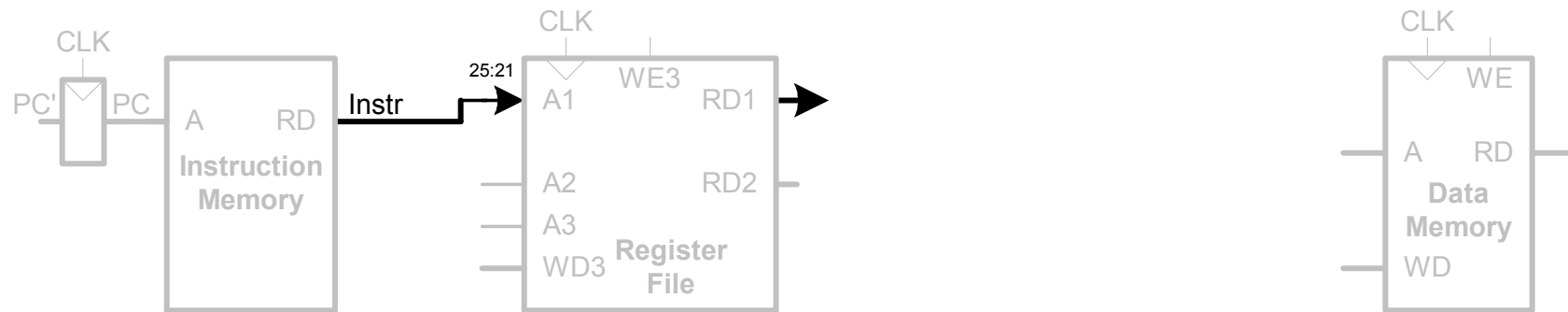
# Single-Cycle Datapath: $1_w$ fetch

## STEP 1: Fetch instruction



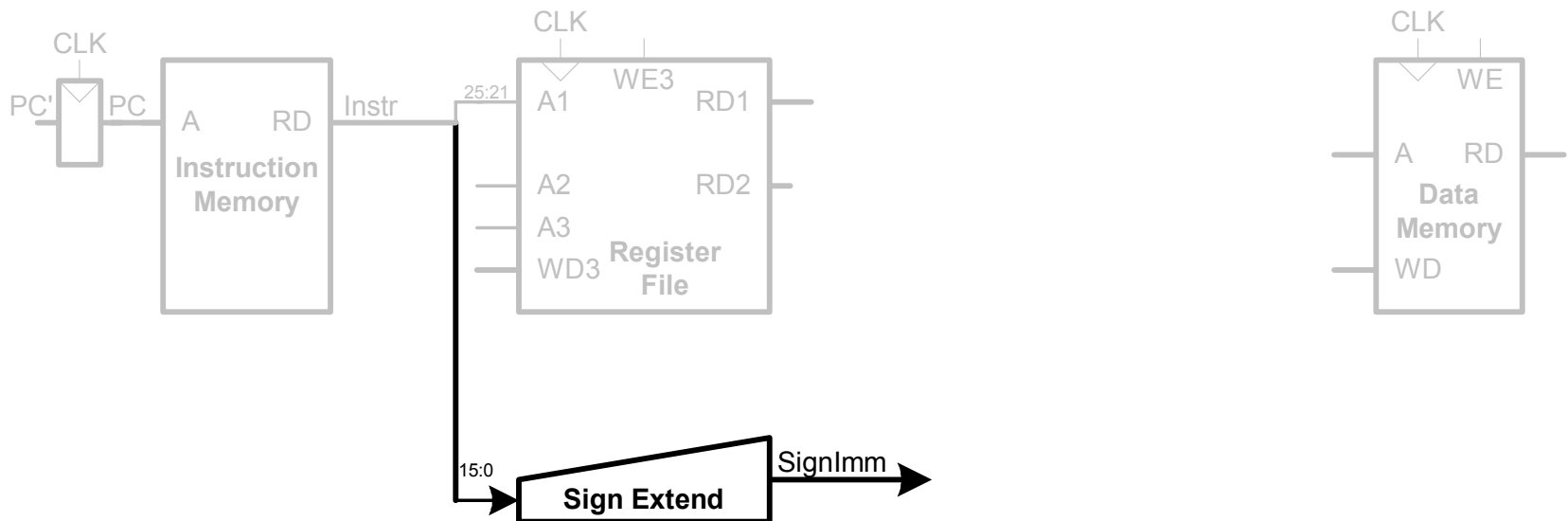
# Single-Cycle Datapath: $1_W$ Register Read

**STEP 2:** Read source operands from RF



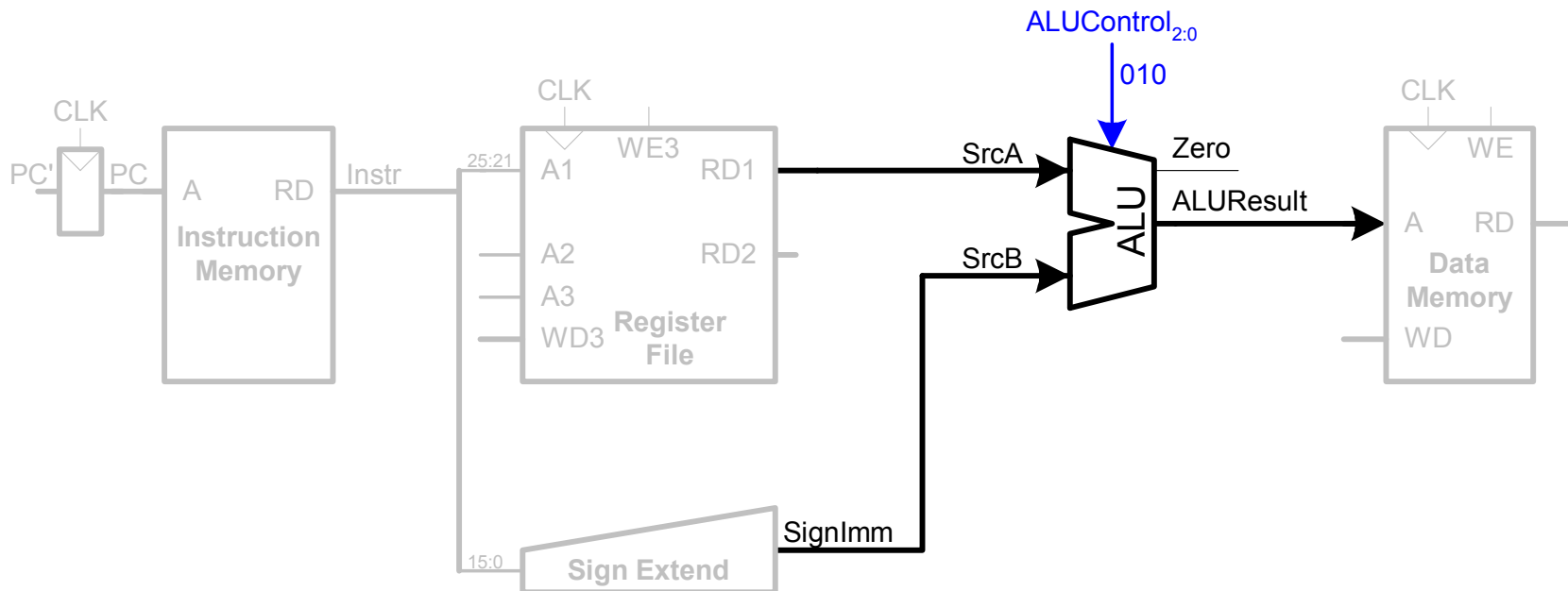
# Single-Cycle Datapath: $1_w$ Immediate

## STEP 3: Sign-extend the immediate



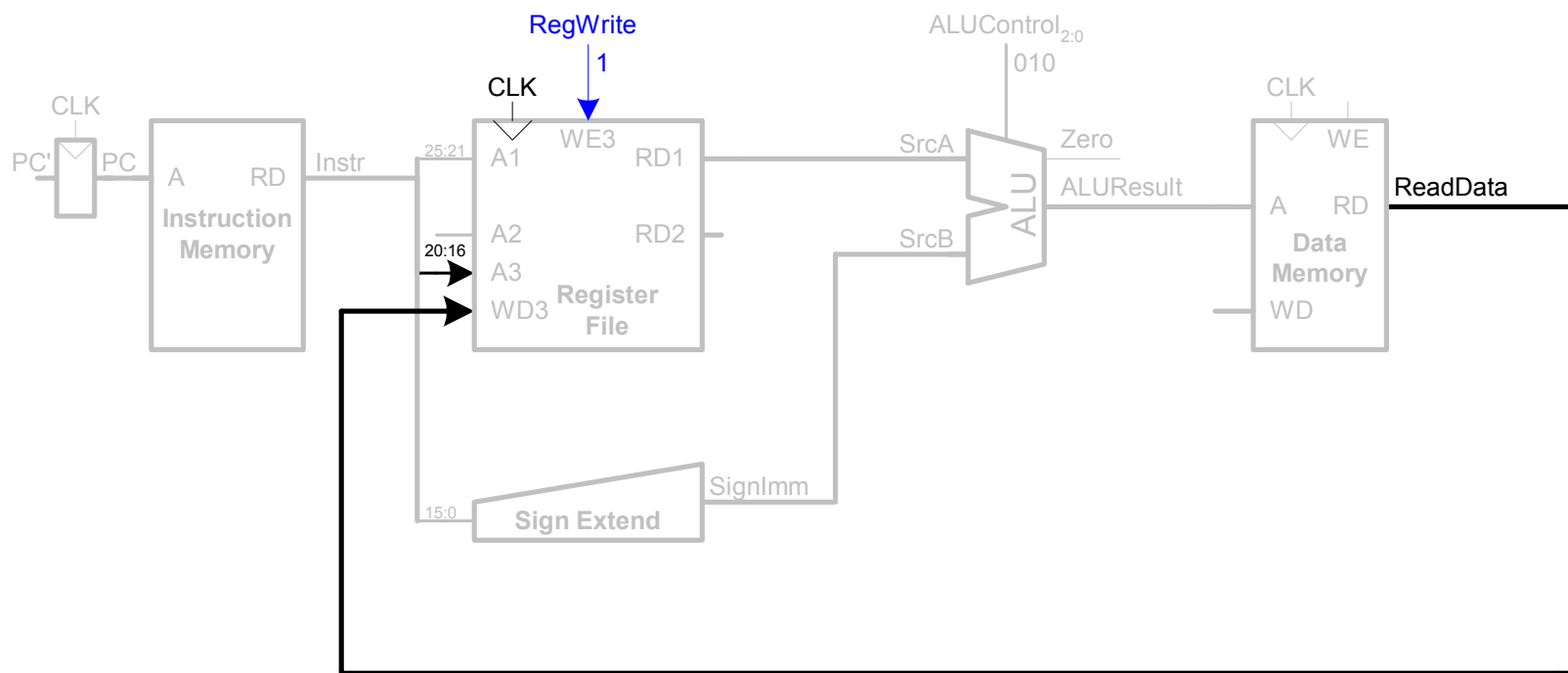
# Single-Cycle Datapath: $lw$ address

## STEP 4: Compute the memory address



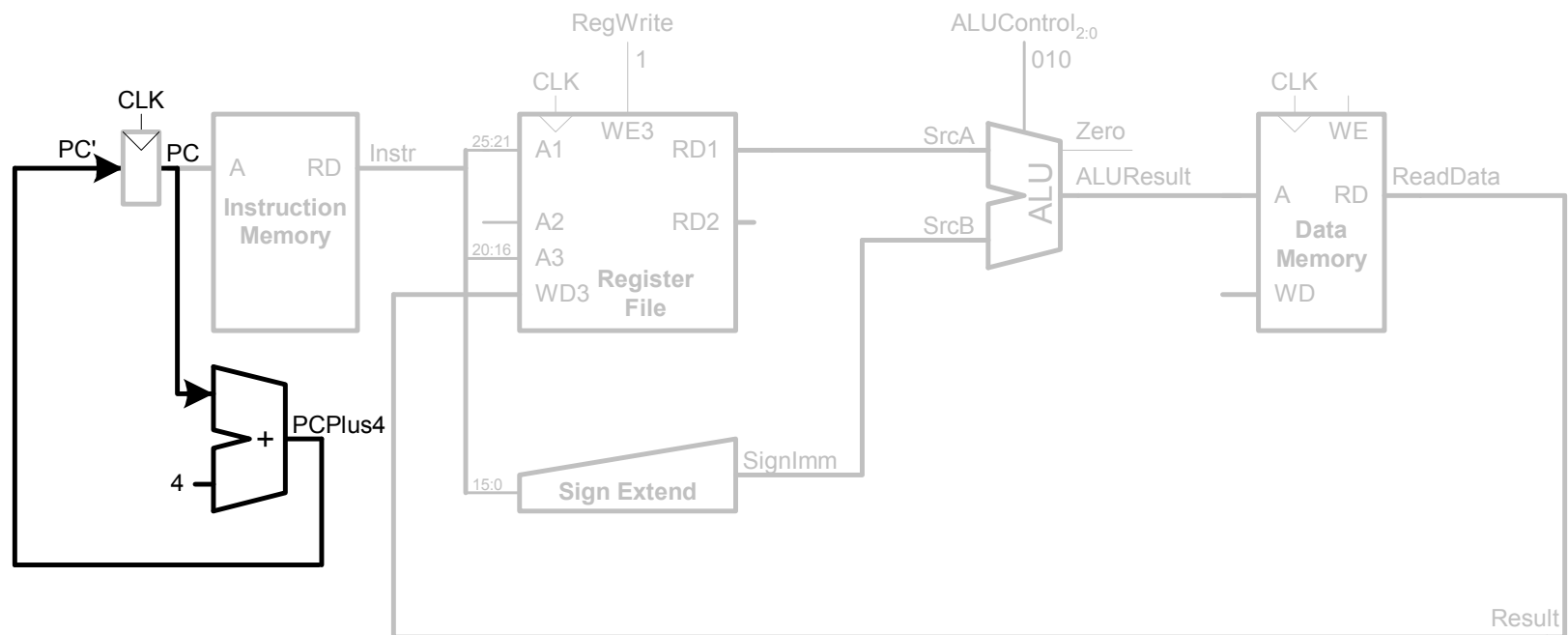
# Single-Cycle Datapath: $1_W$ Memory Read

- STEP 5:** Read data from memory and write it back to register file



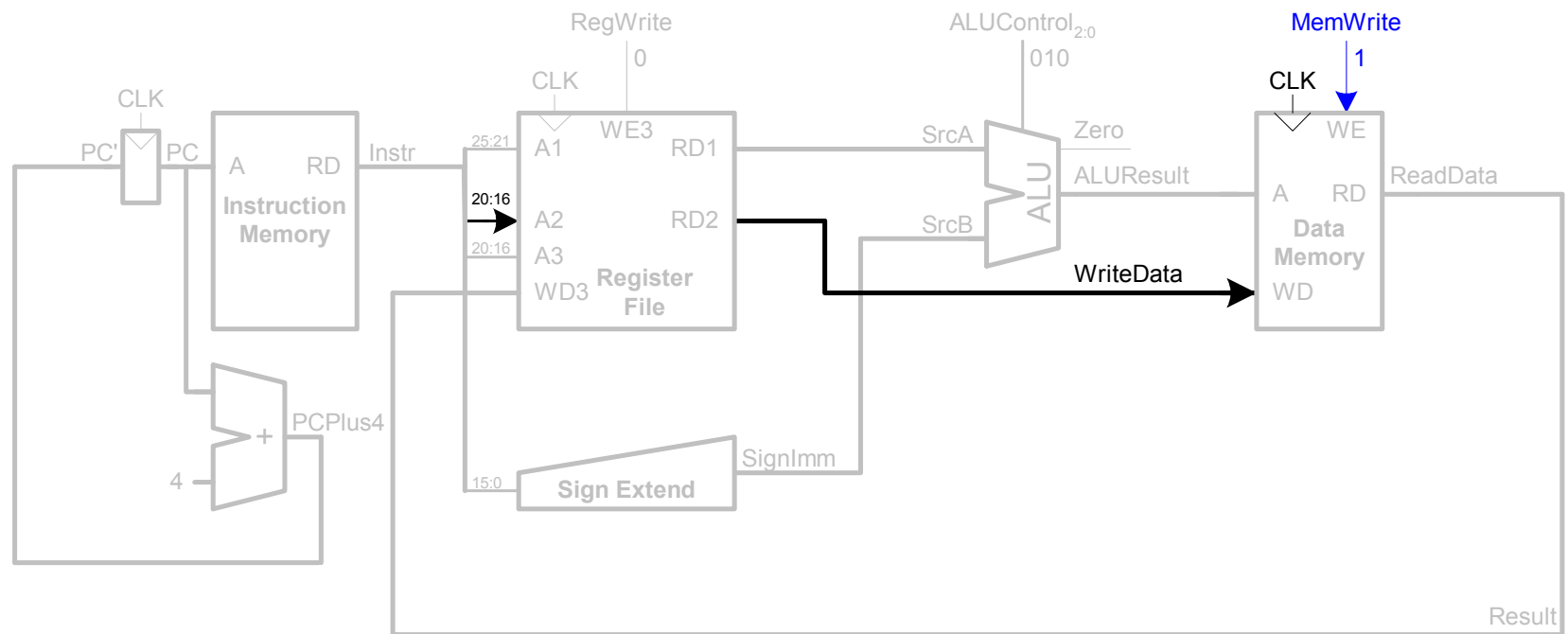
# Single-Cycle Datapath: $1_W$ PC Increment

## STEP 6: Determine address of next instruction



# Single-Cycle Datapath: $sw$

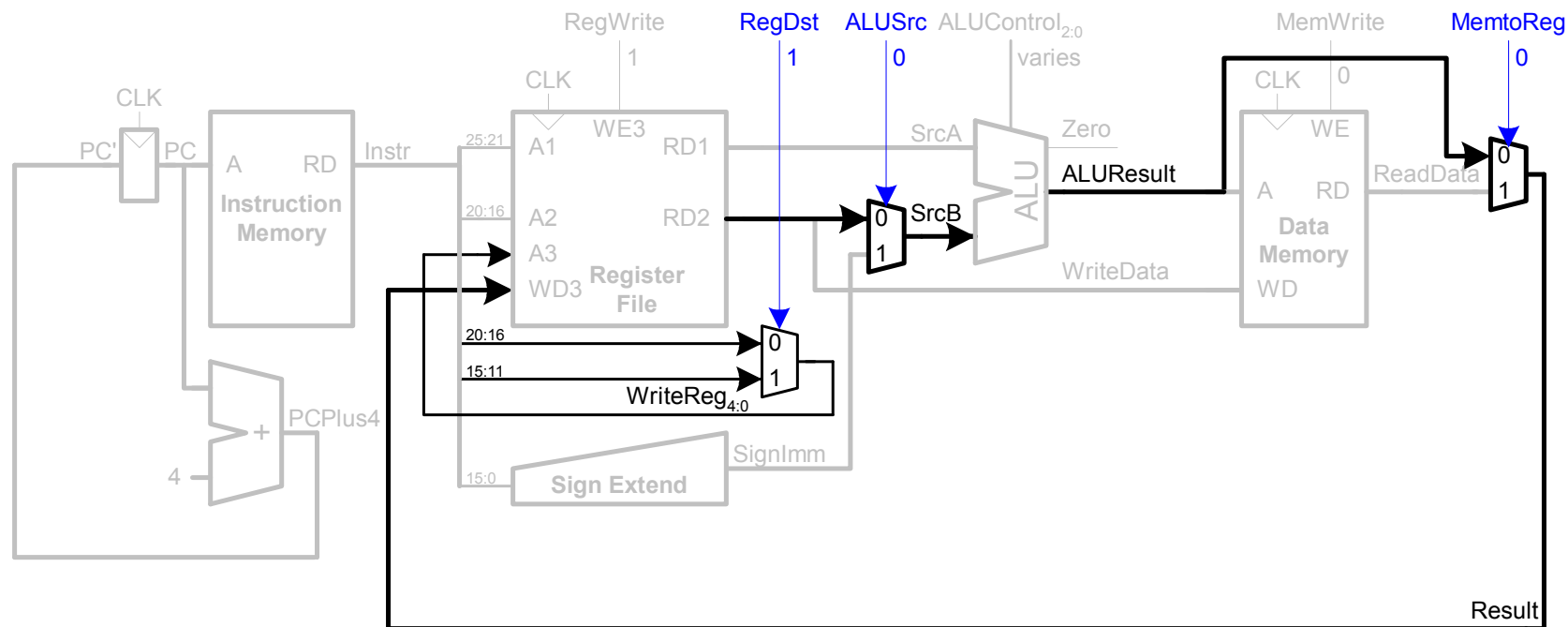
Write data in  $rt$  to memory





# Single-Cycle Datapath: R-Type

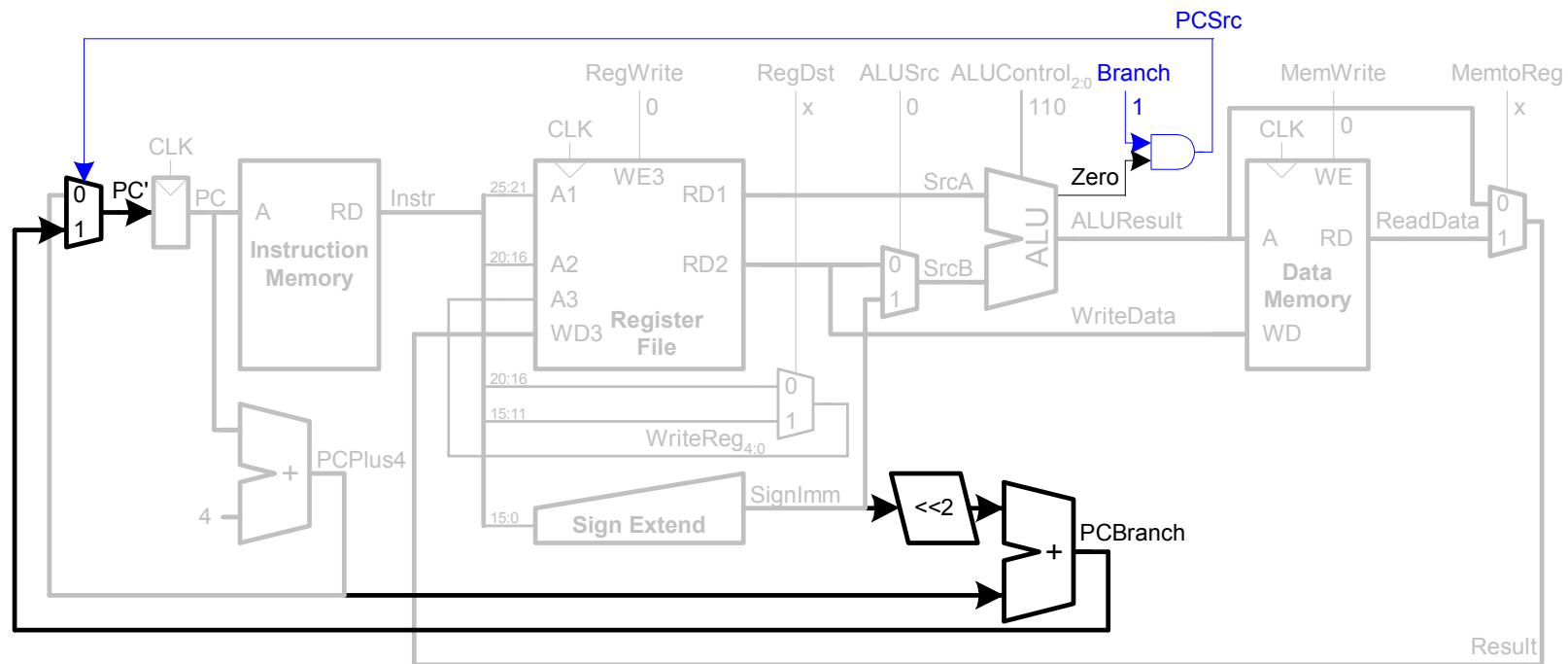
- Read from `rs` and `rt`
- Write `ALUResult` to register file
- Write to `rd` (instead of `rt`)



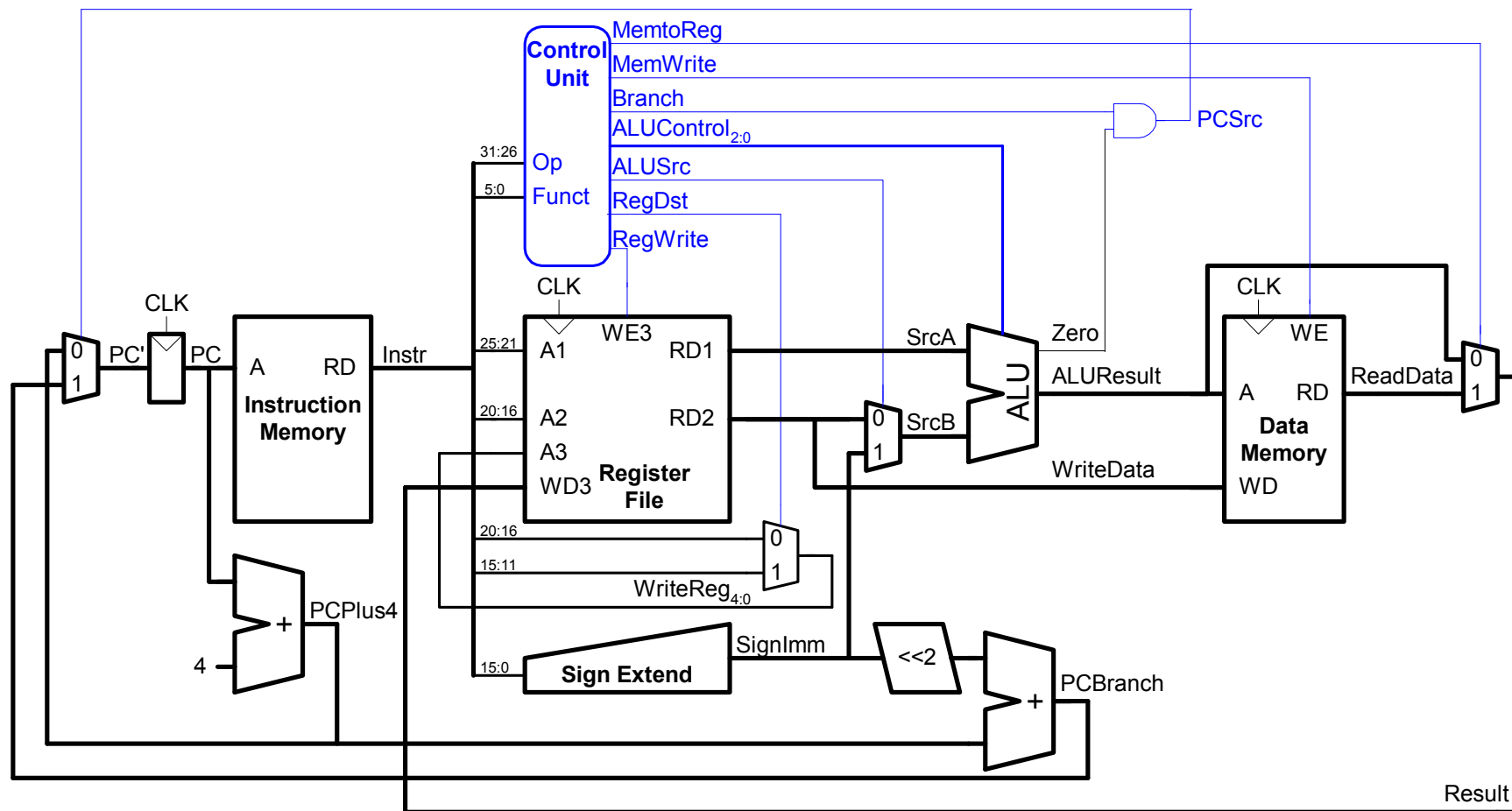
# Single-Cycle Datapath: beq

- Determine whether values in  $rs$  and  $rt$  are equal
- Calculate branch target address:

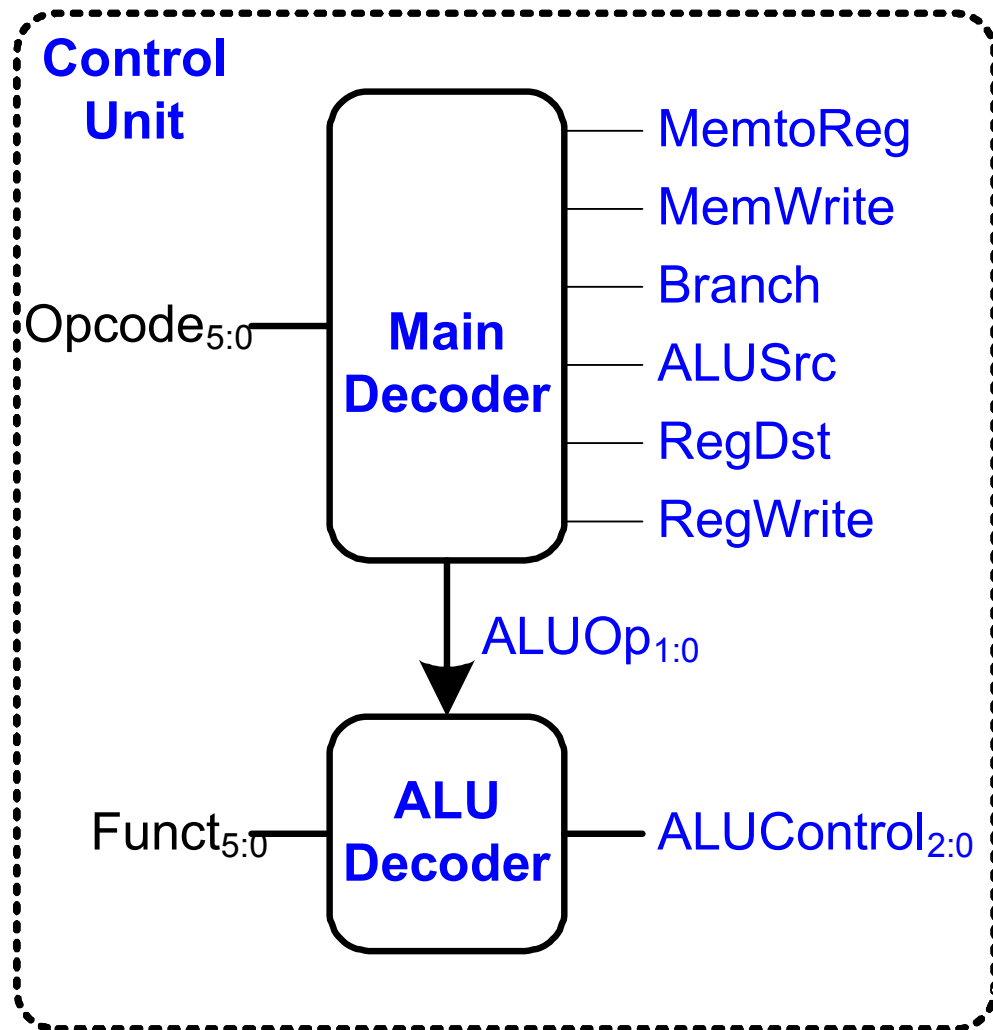
$$BTA = (\text{sign-extended immediate} \ll 2) + (PC+4)$$



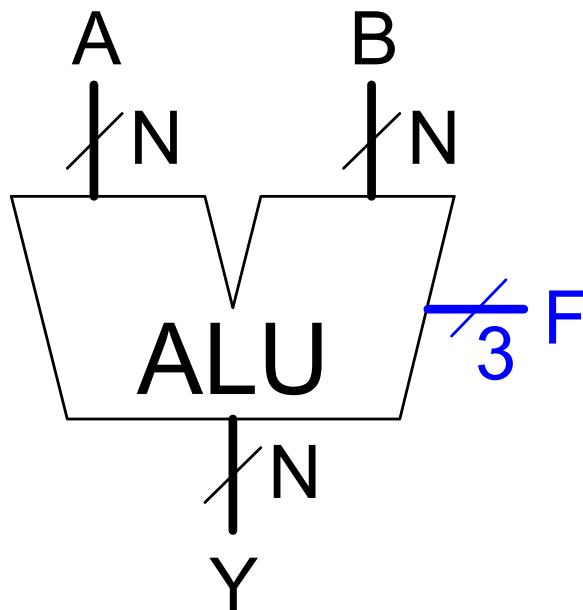
# Single-Cycle Processor



# Single-Cycle Control

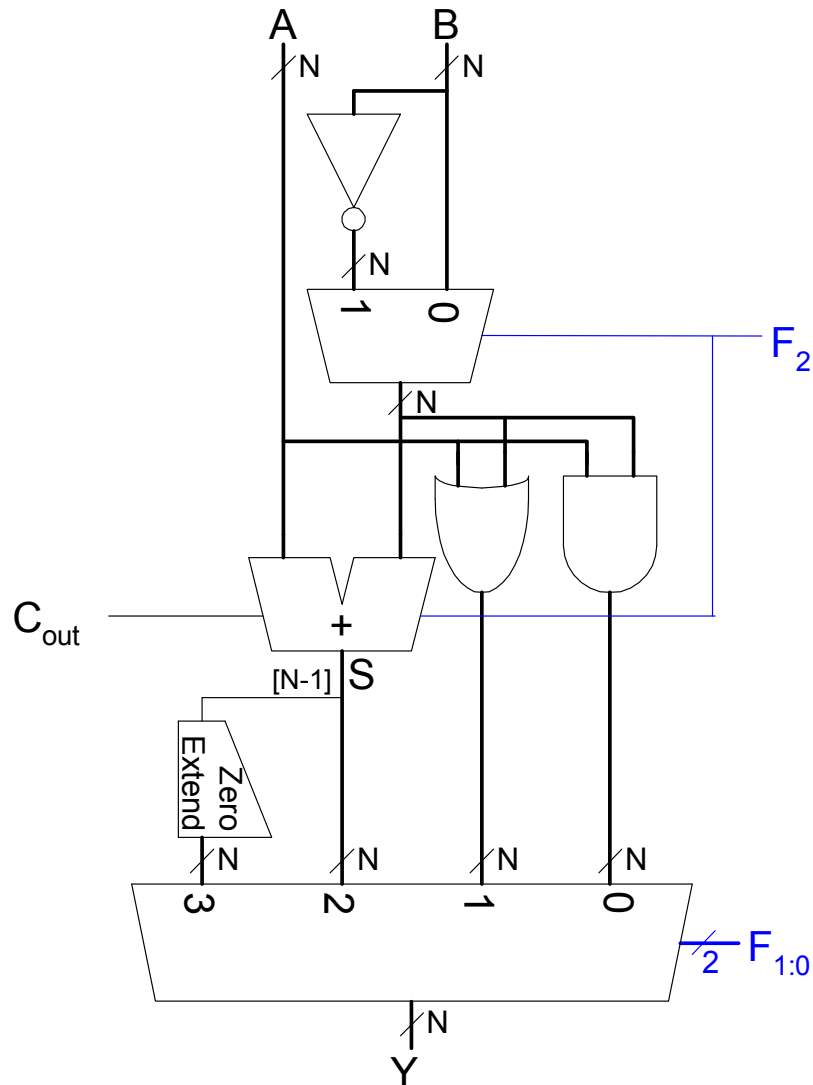


# Review: ALU



$F_{2:0}$	Function
000	$A \& B$
001	$A   B$
010	$A + B$
011	not used
100	$A \& \sim B$
101	$A   \sim B$
110	$A - B$
111	SLT

# Review: ALU



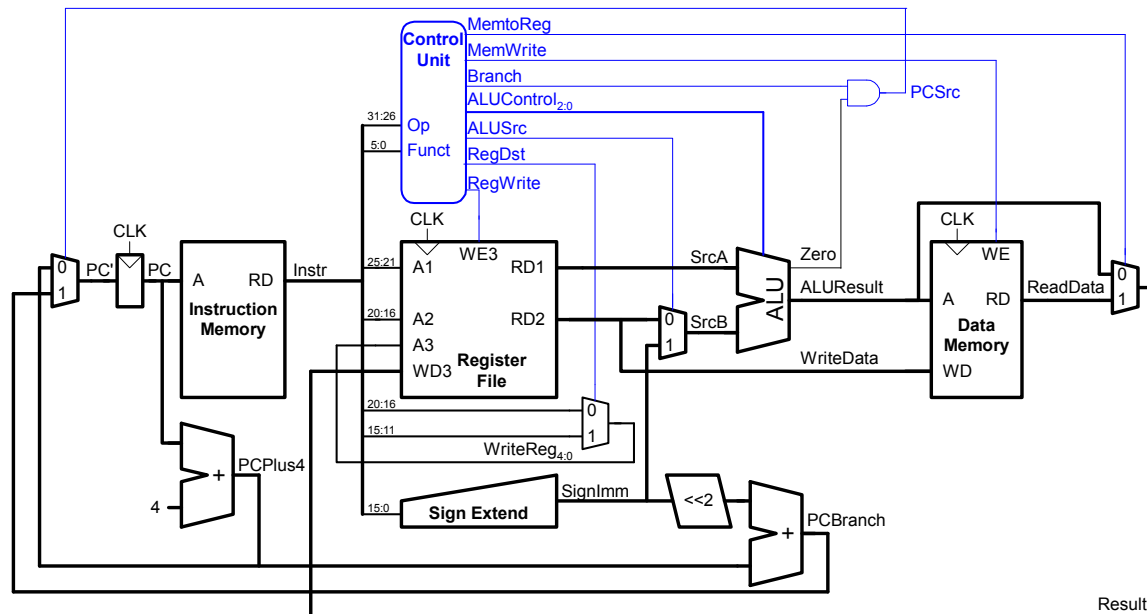
# Control Unit: ALU Decoder

ALUOp <sub>1:0</sub>	Meaning
00	Add
01	Subtract
10	Look at Funct
11	Not Used

ALUOp <sub>1:0</sub>	Funct	ALUControl <sub>2:0</sub>
00	X	010 (Add)
X1	X	110 (Subtract)
1X	100000 (add)	010 (Add)
1X	100010 (sub)	110 (Subtract)
1X	100100 (and)	000 (And)
1X	100101 (or)	001 (Or)
1X	101010 (slt)	111 (SLT)

# Control Unit Main Decoder

Instruction	Op <sub>5:0</sub>	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	ALUOp <sub>1:0</sub>
R-type	000000							
lw	100011							
sw	101011							
beq	000100							

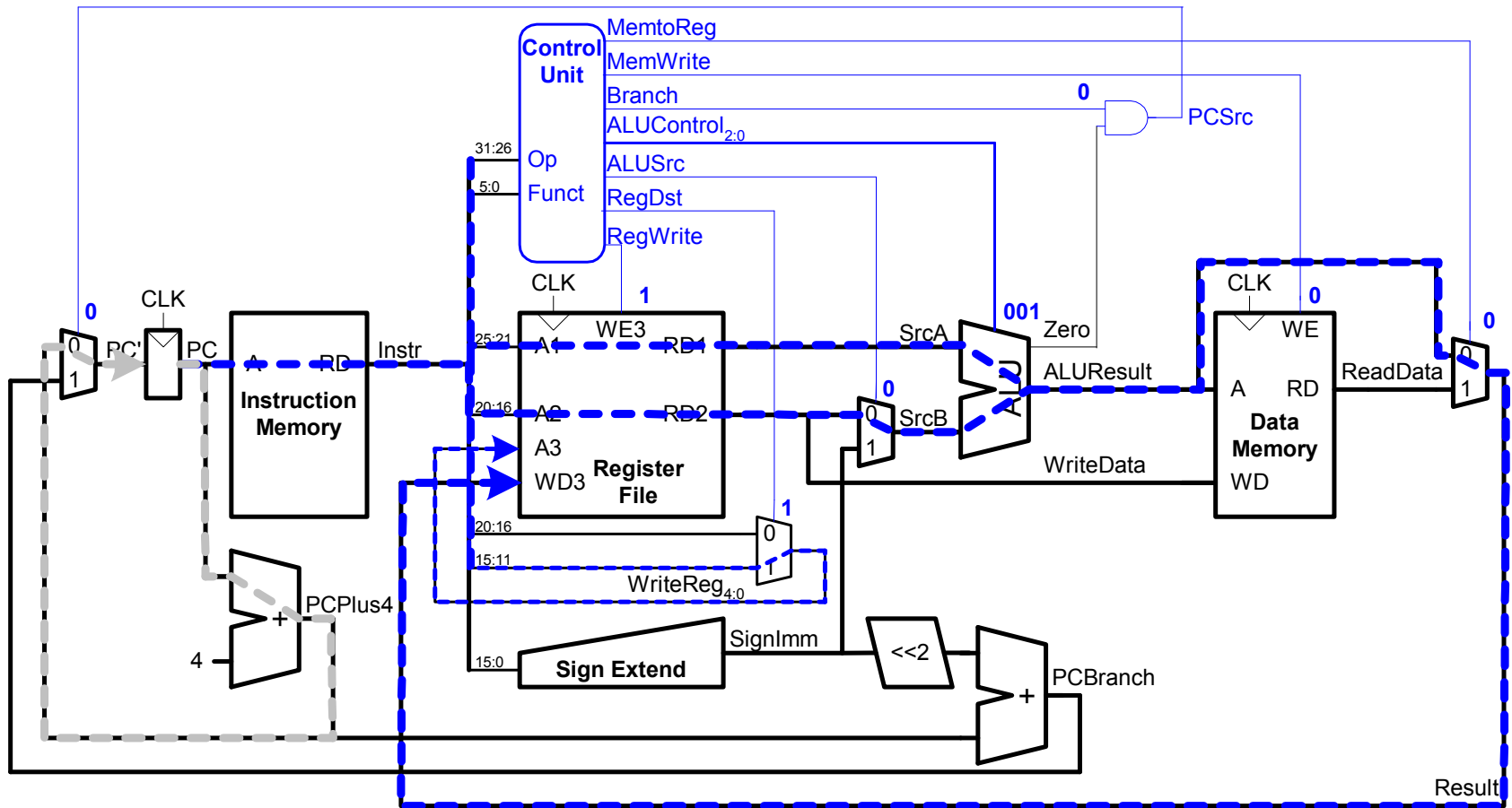




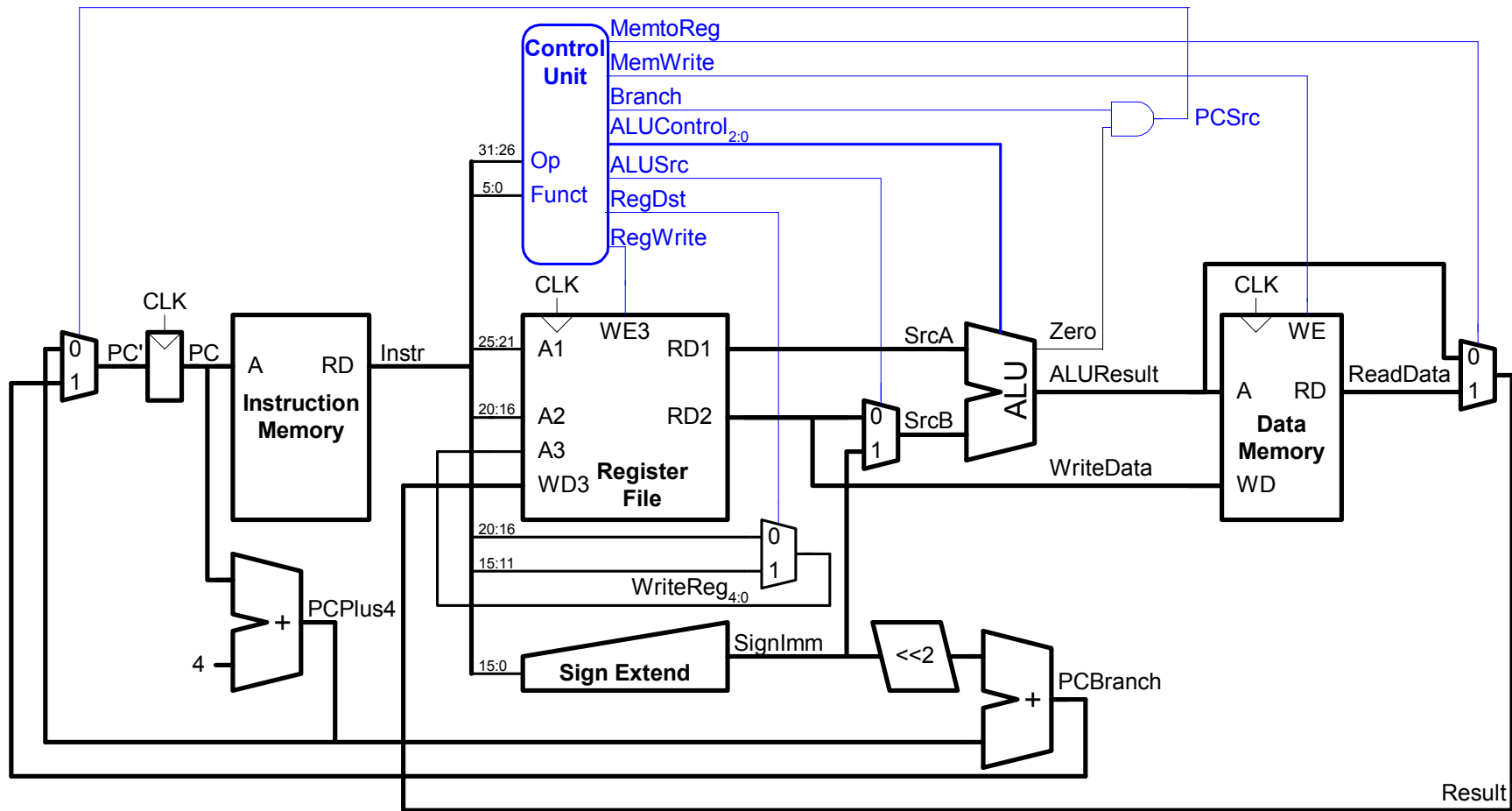
# Control Unit: Main Decoder

Instruction	Op <sub>5:0</sub>	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	ALUOp <sub>1:0</sub>
R-type	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	0	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01

# Single-Cycle Datapath: or



# Extended Functionality: addi



No change to datapath

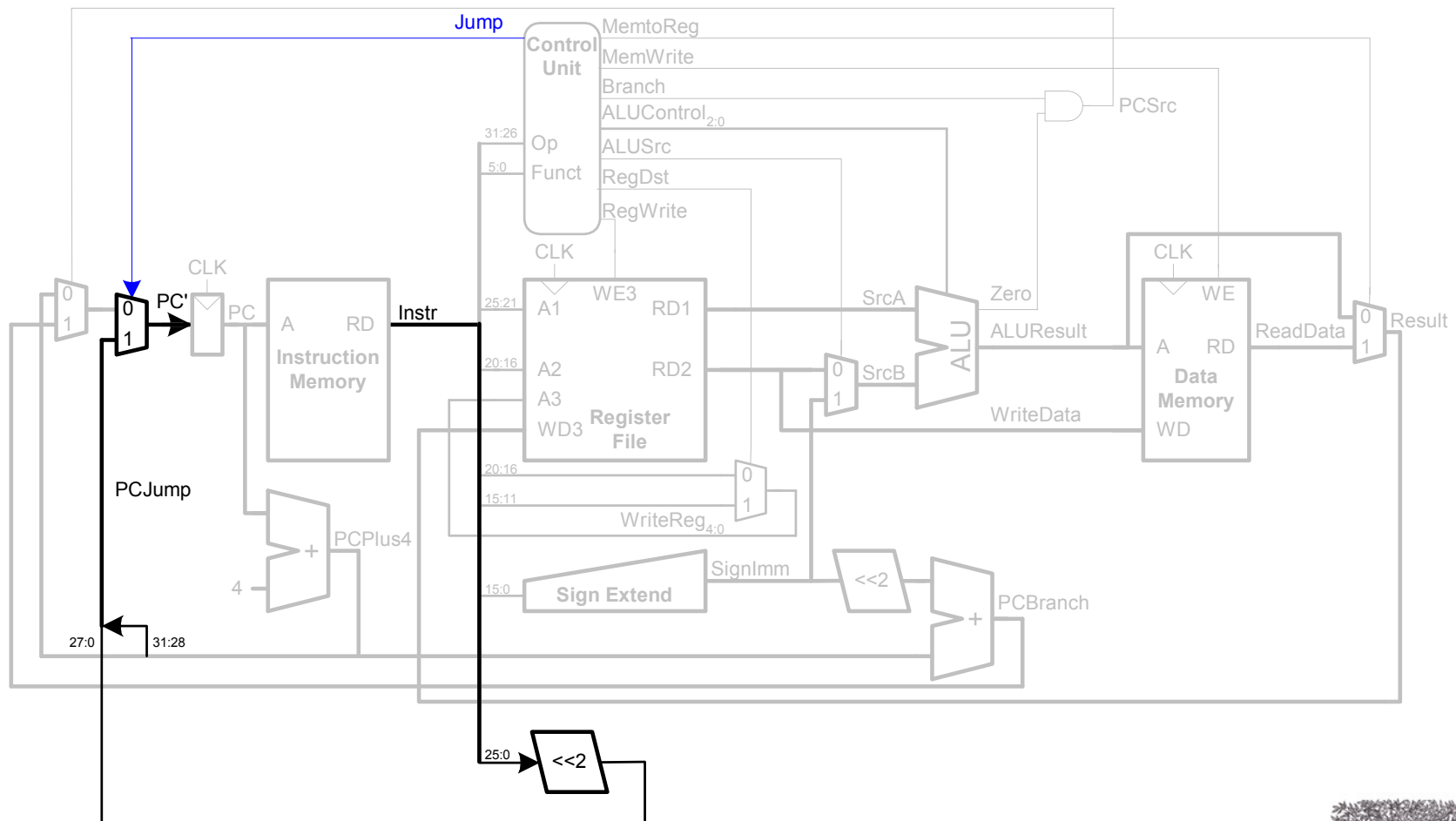
## Control Unit: addi

Instruction	Op <sub>5:0</sub>	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	ALUOp <sub>1:0</sub>
R-type	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01
<b>addi</b>	<b>001000</b>							

## Control Unit: addi

Instruction	Op <sub>5:0</sub>	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	ALUOp <sub>1:0</sub>
R-type	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01
<b>addi</b>	<b>001000</b>	<b>1</b>	<b>0</b>	<b>1</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>00</b>

# Extended Functionality: j



# Control Unit: Main Decoder

Instruction	Op <sub>5:0</sub>	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	ALUOp <sub>1:0</sub>	Jump
R-type	000000	1	1	0	0	0	0	10	0
lw	100011	1	0	1	0	0	1	00	0
sw	101011	0	X	1	0	1	X	00	0
beq	000100	0	X	0	1	0	X	01	0
j	000010								

# Control Unit: Main Decoder

Instruction	Op <sub>5:0</sub>	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	ALUOp <sub>1:0</sub>	Jump
R-type	000000	1	1	0	0	0	0	10	0
lw	100011	1	0	1	0	0	1	00	0
sw	101011	0	X	1	0	1	X	00	0
beq	000100	0	X	0	1	0	X	01	0
j	000010	0	X	X	X	0	X	XX	1



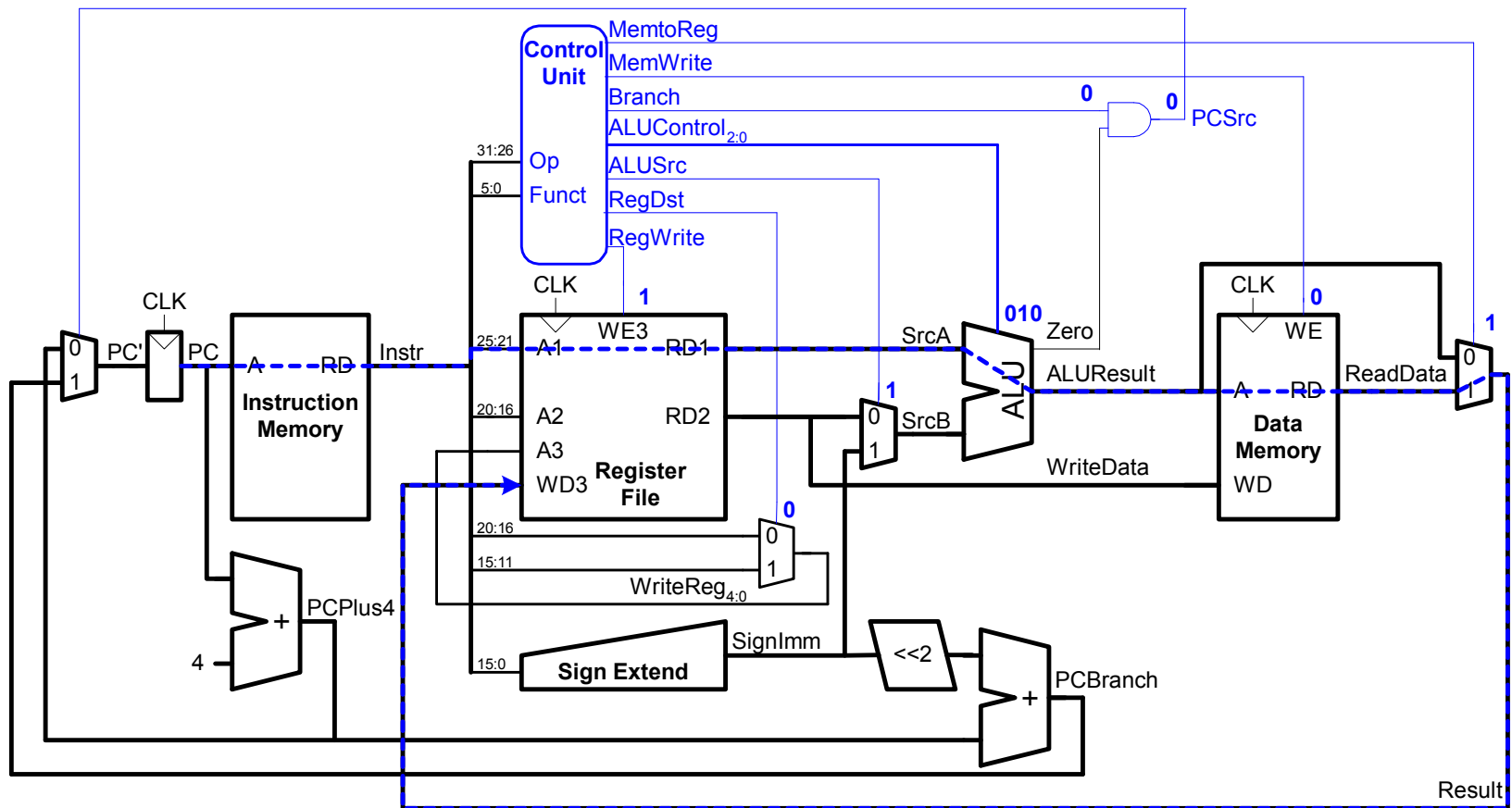
# Review: Processor Performance

## Program Execution Time

$$= (\text{\#instructions})(\text{cycles/instruction})(\text{seconds/cycle})$$

$$= \text{\# instructions} \times \text{CPI} \times T_c$$

# Single-Cycle Performance



$T_c$  limited by critical path (1w)

# Single-Cycle Performance

- Single-cycle critical path:

$$T_c = t_{pcq\_PC} + t_{mem} + \max(t_{RFread}, t_{sext} + t_{mux}) + t_{ALU} + t_{mem} + t_{mux} + t_{RFsetup}$$

- Typically, limiting paths are:

- memory, ALU, register file

- $T_c = t_{pcq\_PC} + 2t_{mem} + t_{RFread} + t_{mux} + t_{ALU} + t_{RFsetup}$



# Single-Cycle Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	$t_{pcq\_PC}$	30
Register setup	$t_{setup}$	20
Multiplexer	$t_{mux}$	25
ALU	$t_{ALU}$	200
Memory read	$t_{mem}$	250
Register file read	$t_{RFread}$	150
Register file setup	$t_{RFsetup}$	20

$$T_c = ?$$

# Single-Cycle Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	$t_{pcq\_PC}$	30
Register setup	$t_{setup}$	20
Multiplexer	$t_{mux}$	25
ALU	$t_{ALU}$	200
Memory read	$t_{mem}$	250
Register file read	$t_{RFread}$	150
Register file setup	$t_{RFsetup}$	20

$$\begin{aligned}
 T_c &= t_{pcq\_PC} + 2t_{mem} + t_{RFread} + t_{mux} + t_{ALU} + t_{RFsetup} \\
 &= [30 + 2(250) + 150 + 25 + 200 + 20] \text{ ps} \\
 &= 925 \text{ ps}
 \end{aligned}$$

# Single-Cycle Performance Example

Program with 100 billion instructions:

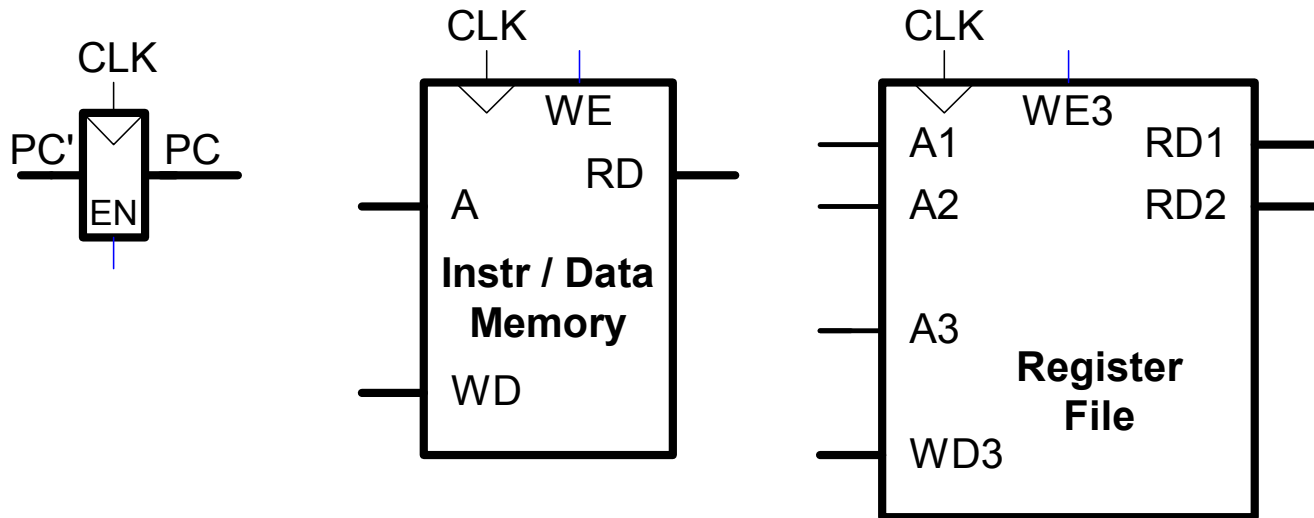
$$\begin{aligned}\text{Execution Time} &= \# \text{ instructions} \times \text{CPI} \times T_C \\ &= (100 \times 10^9)(1)(925 \times 10^{-12} \text{ s}) \\ &= \mathbf{92.5 \text{ seconds}}\end{aligned}$$

# Multicycle MIPS Processor

- **Single-cycle:**
  - + simple
  - cycle time limited by longest instruction ( $1_w$ )
  - 2 adders/ALUs & 2 memories
- **Multicycle:**
  - + higher clock speed
  - + simpler instructions run faster
  - + reuse expensive hardware on multiple cycles
  - sequencing overhead paid many times
- **Same design steps: datapath & control**

# Multicycle State Elements

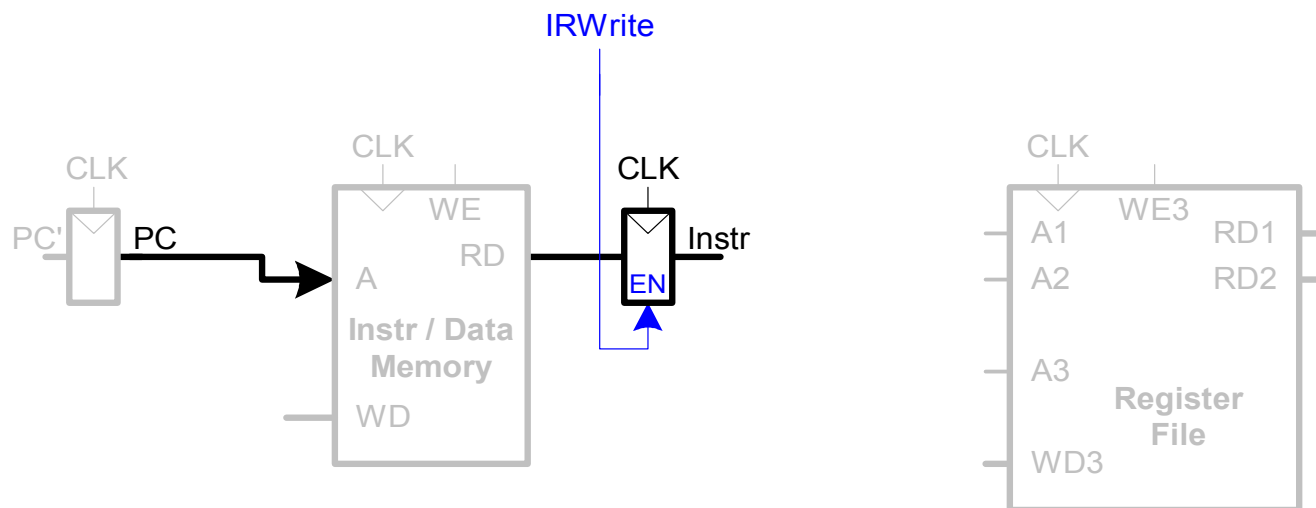
- Replace Instruction and Data memories with a single unified memory – more realistic





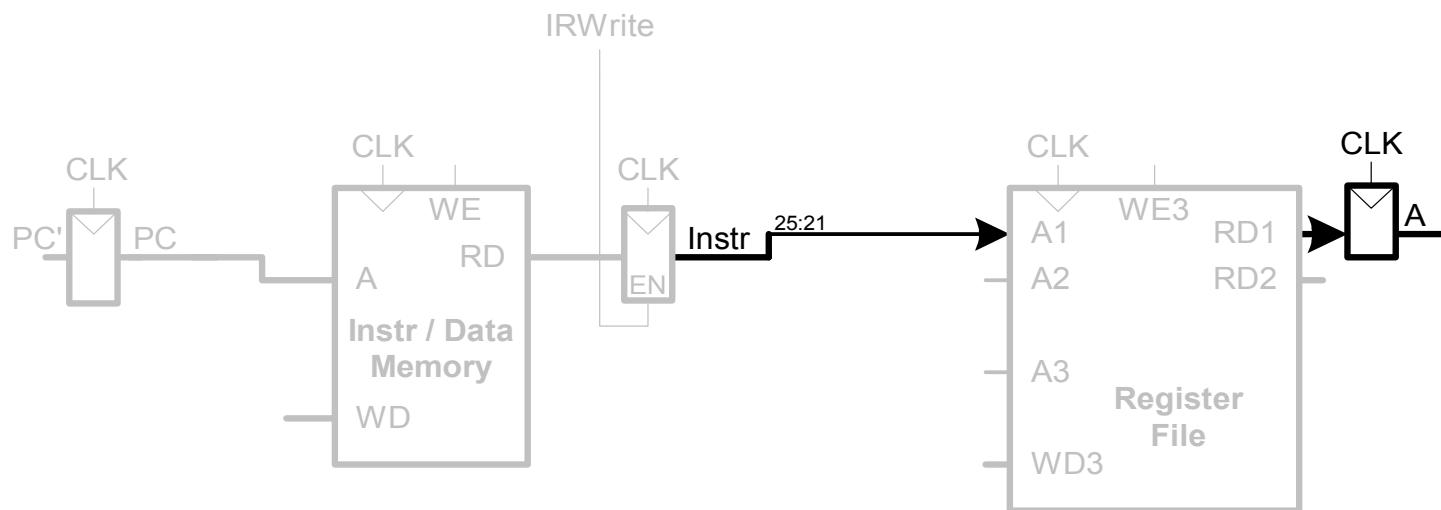
# Multicycle Datapath: Instruction Fetch

## STEP 1: Fetch instruction



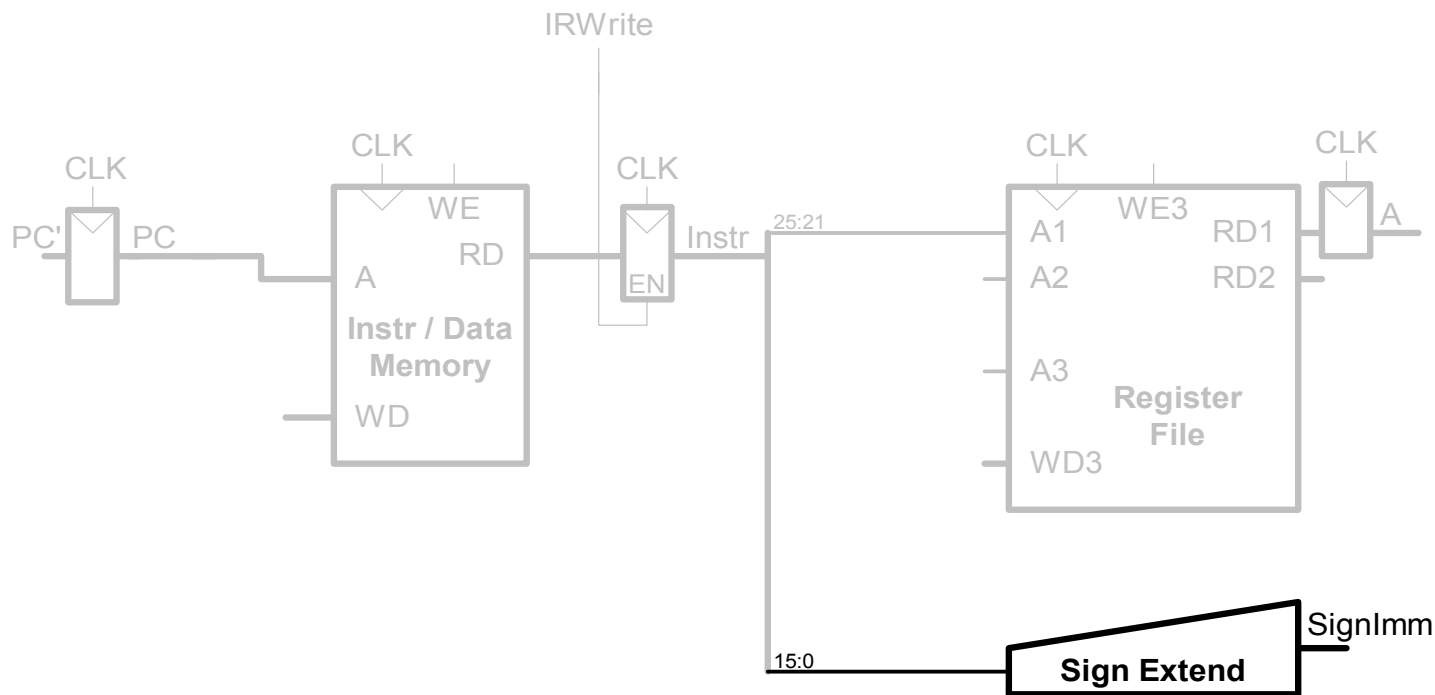
# Multicycle Datapath: $1_w$ Register Read

**STEP 2a:** Read source operands from RF



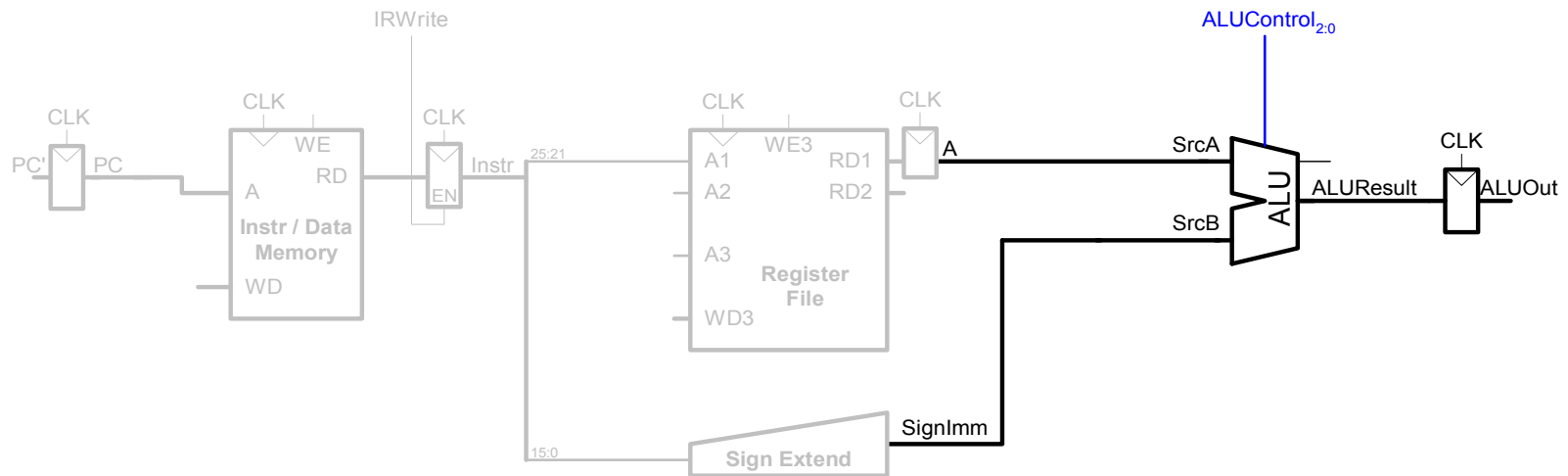
# Multicycle Datapath: $1_w$ Immediate

## STEP 2b: Sign-extend the immediate



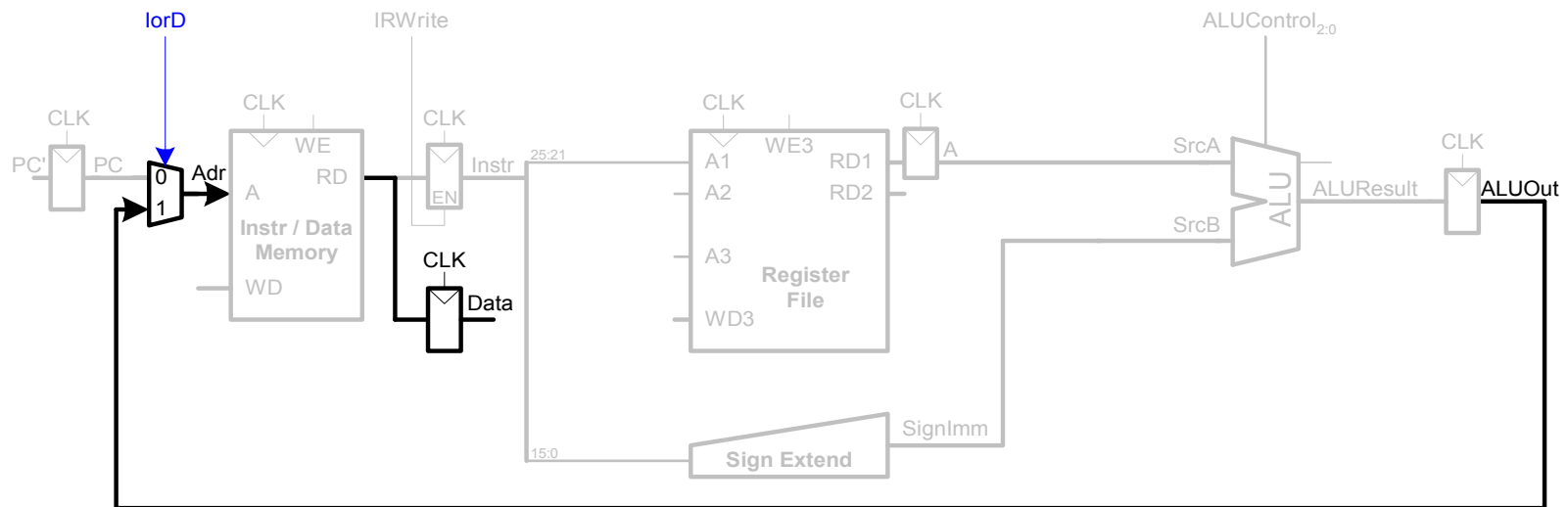
# Multicycle Datapath: $1_W$ Address

## STEP 3: Compute the memory address



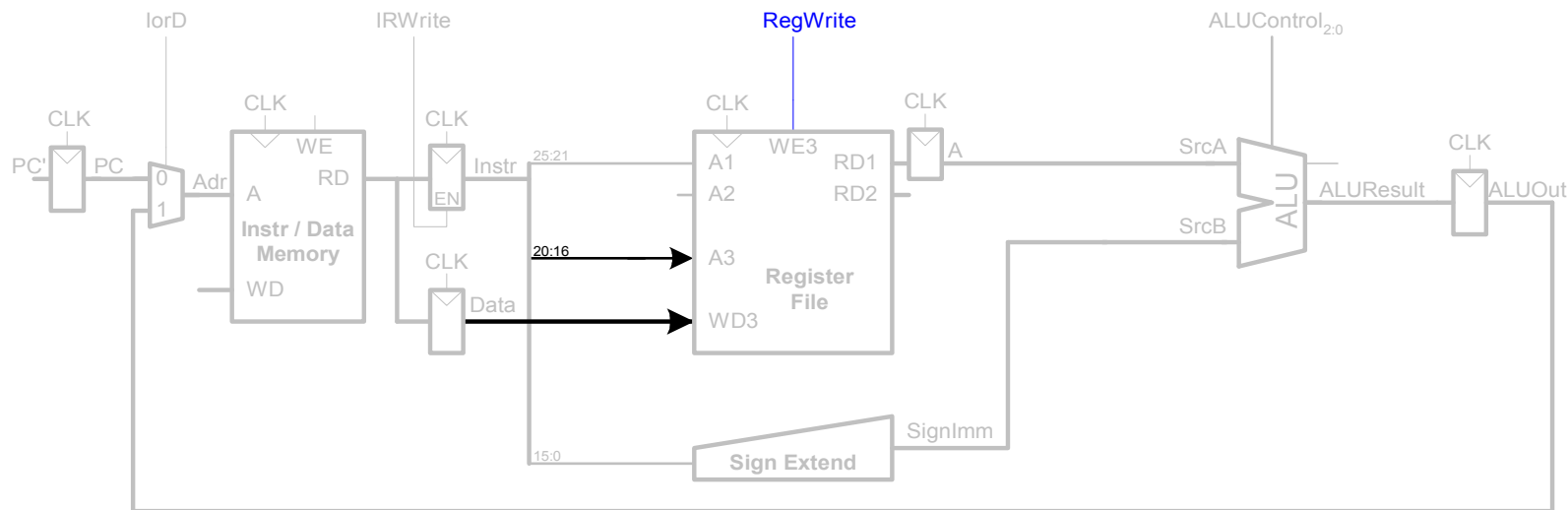
# Multicycle Datapath: $1_W$ Memory Read

## STEP 4: Read data from memory



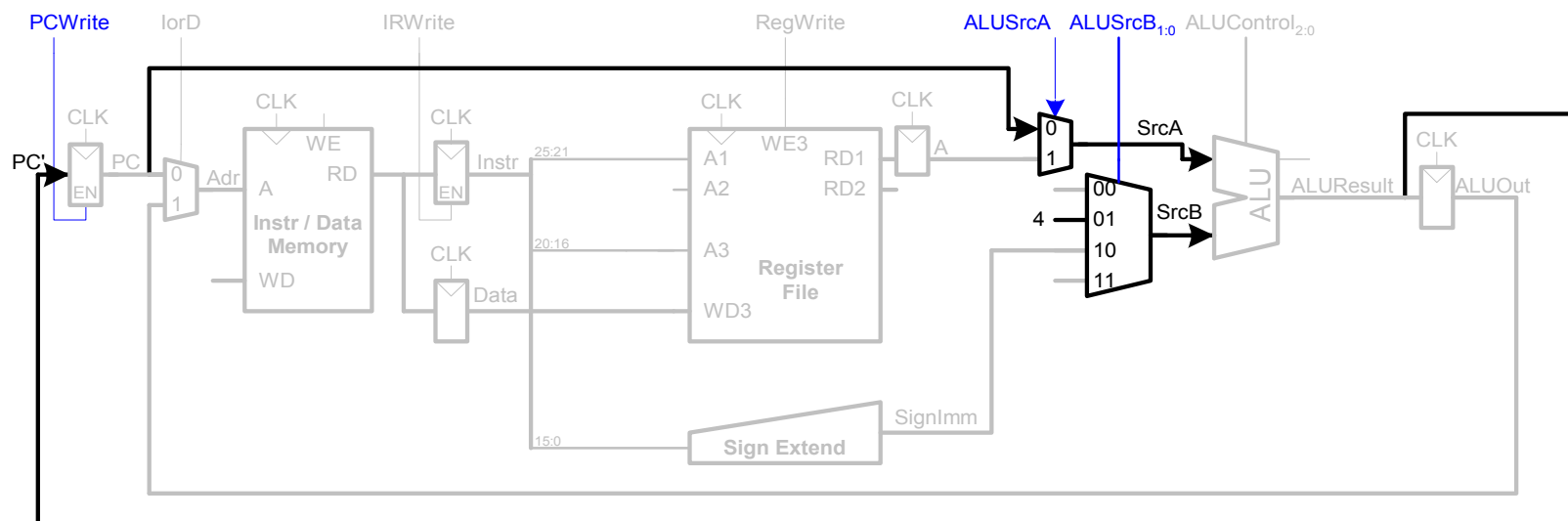
# Multicycle Datapath: $1_W$ Write Register

## STEP 5: Write data back to register file



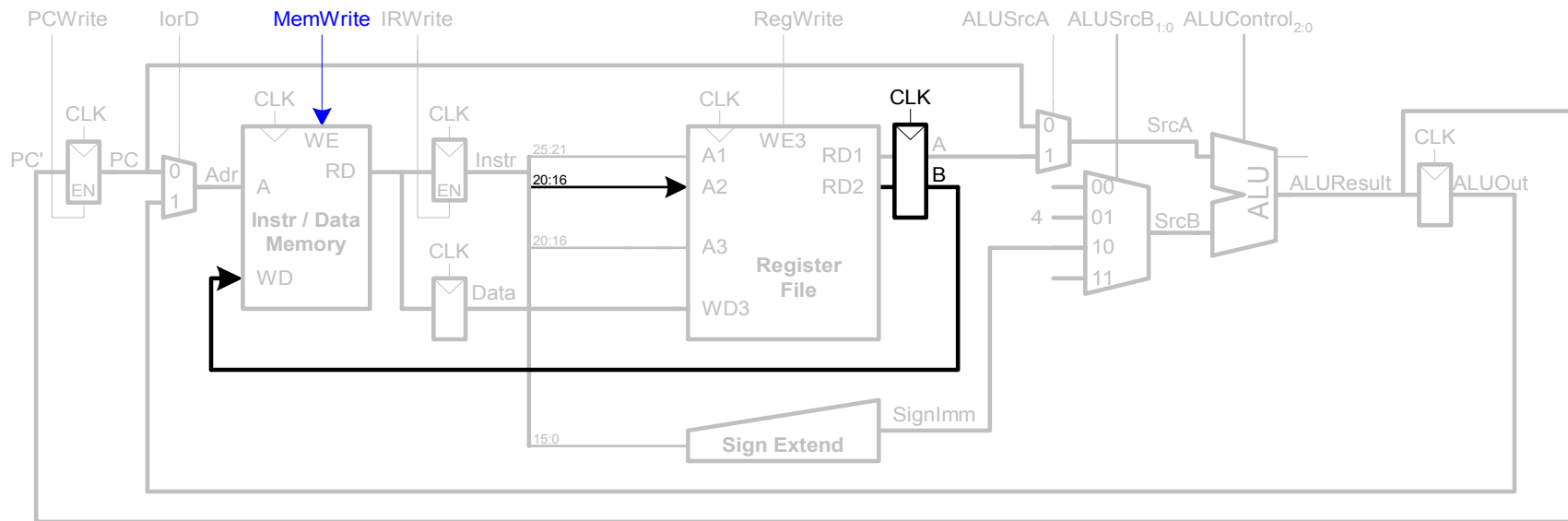
# Multicycle Datapath: Increment PC

## STEP 6: Increment PC



# Multicycle Datapath: $sw$

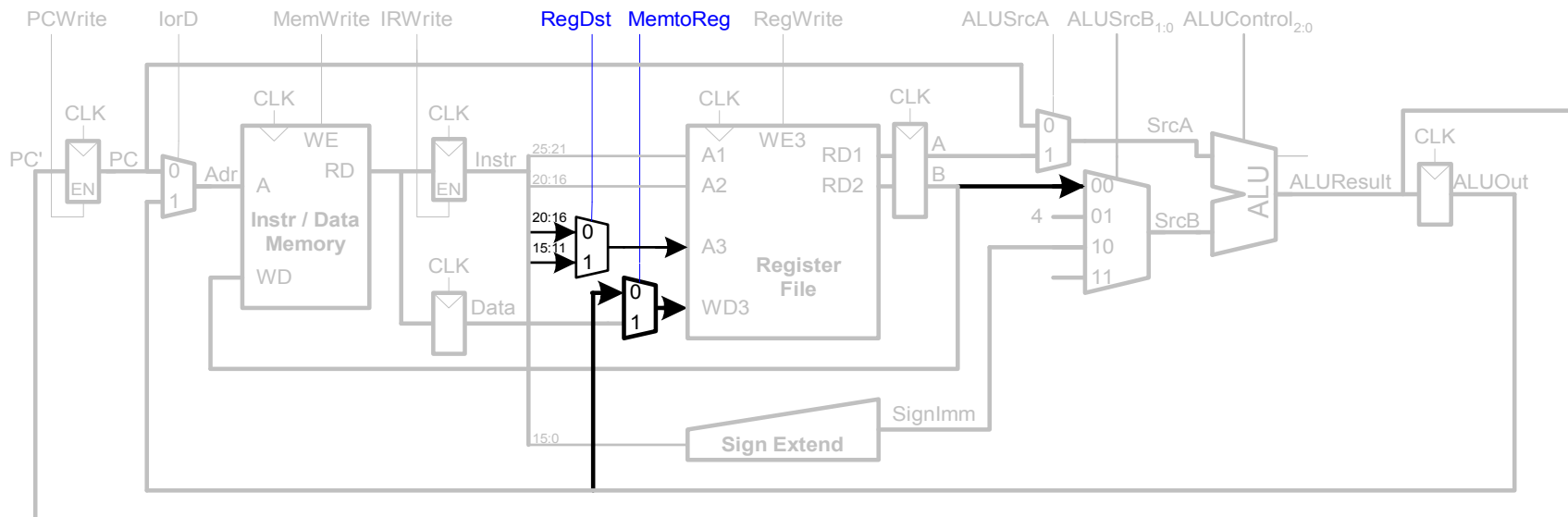
Write data in  $rt$  to memory





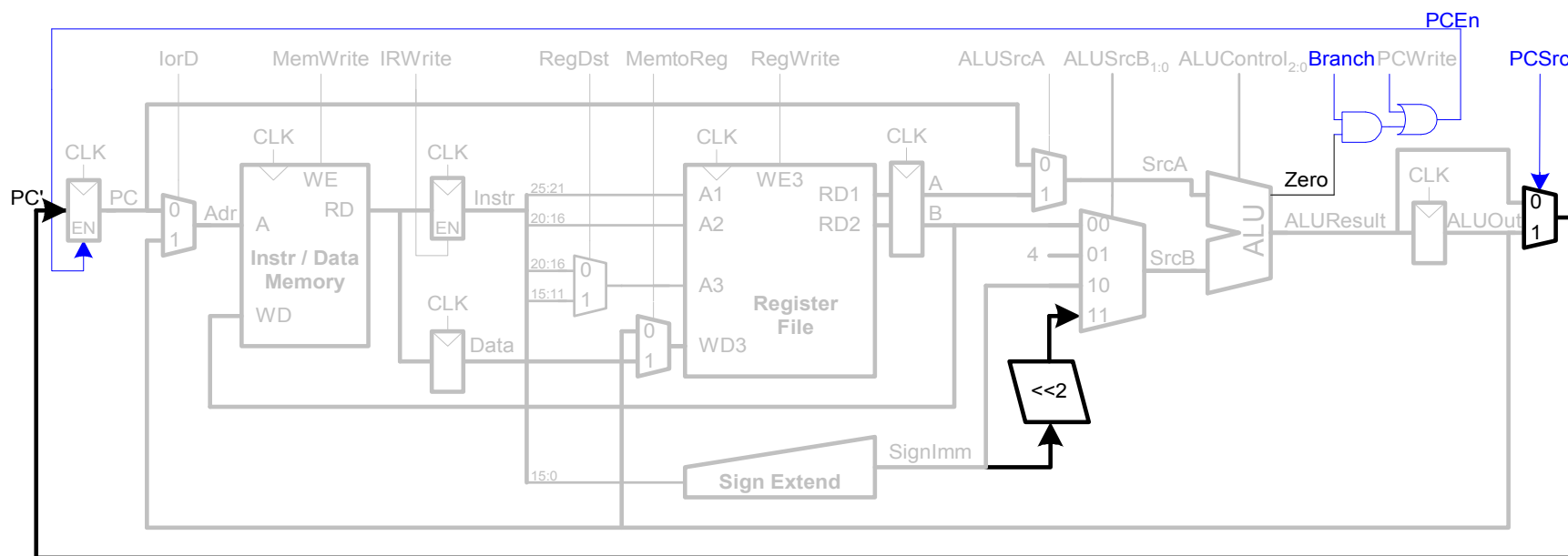
# Multicycle Datapath: R-Type

- Read from  $rs$  and  $rt$
- Write  $ALUResult$  to register file
- Write to  $rd$  (instead of  $rt$ )

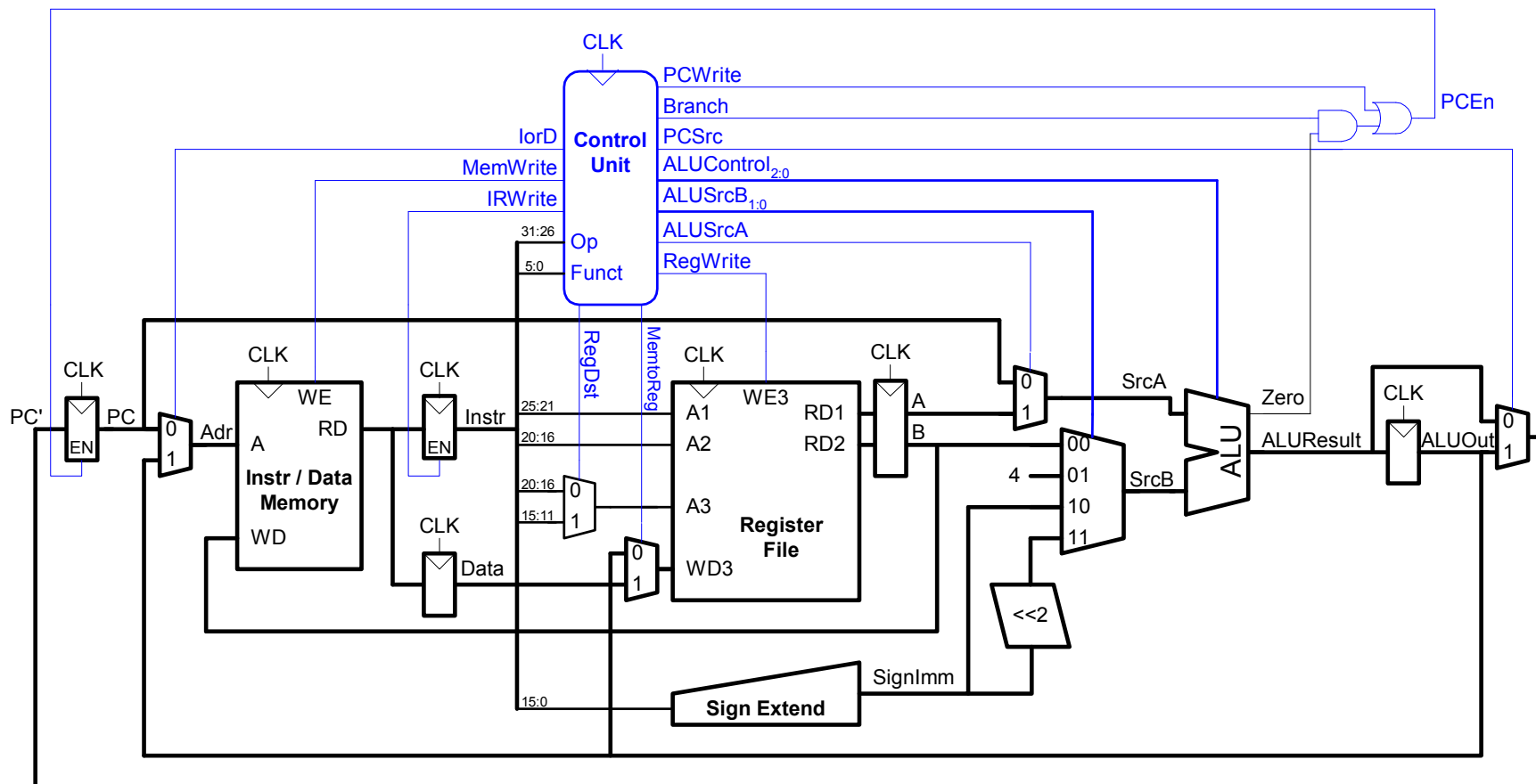


# Multicycle Datapath: beq

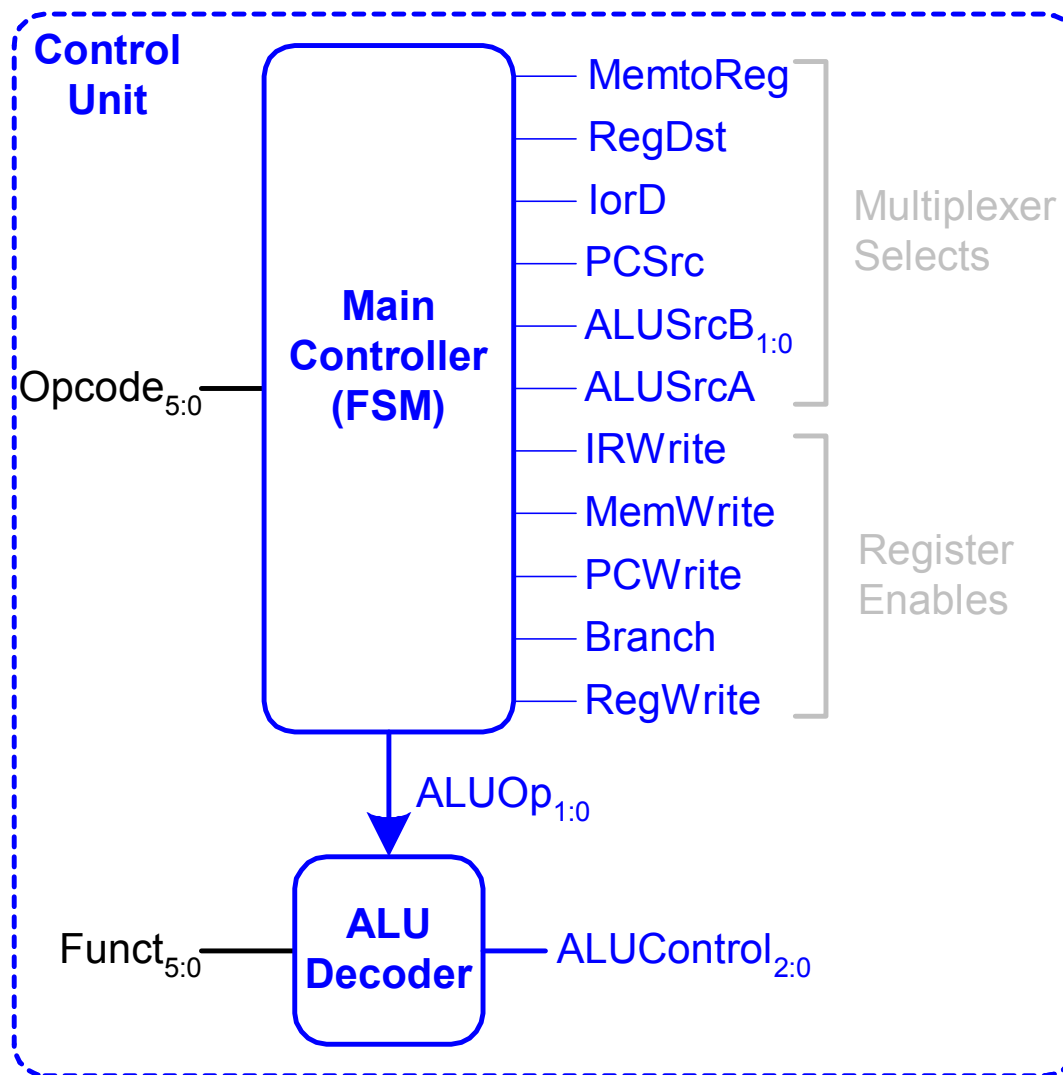
- $rs == rt?$
- $BTA = (\text{sign-extended immediate} \ll 2) + (PC+4)$



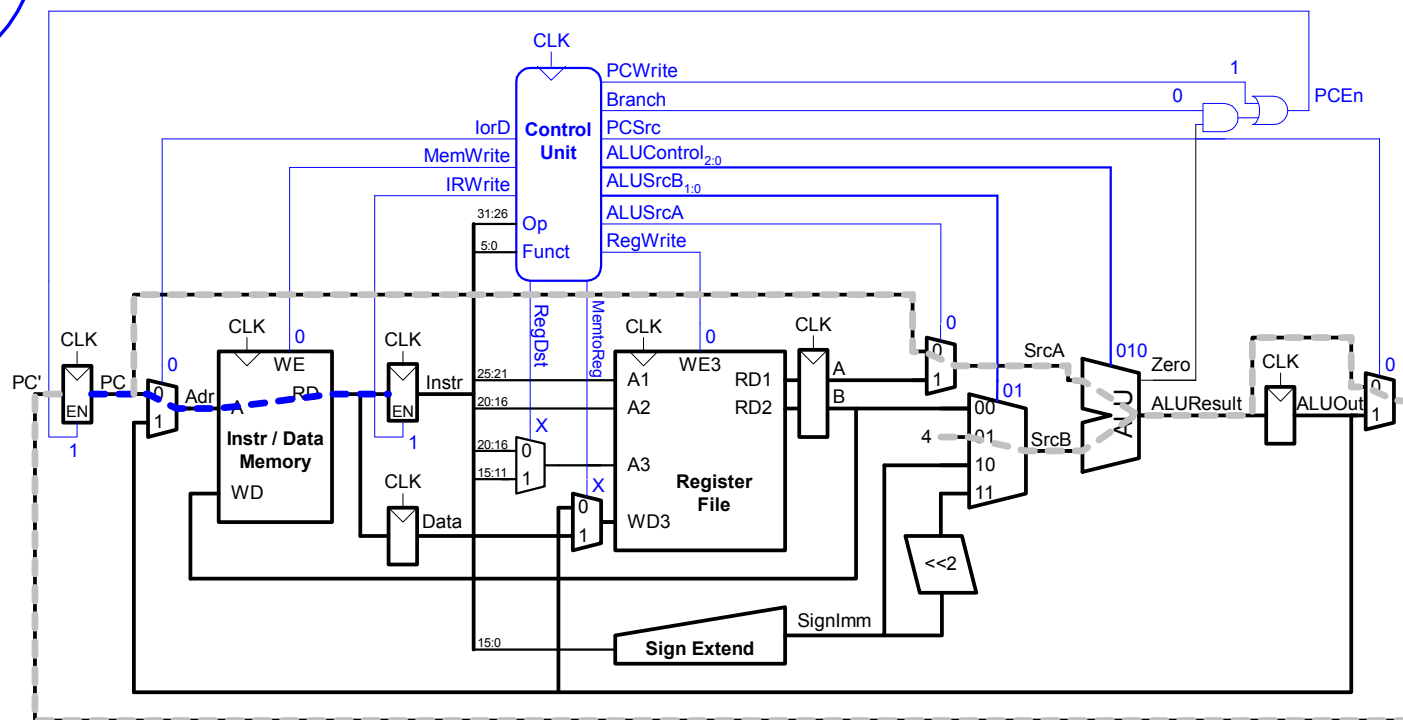
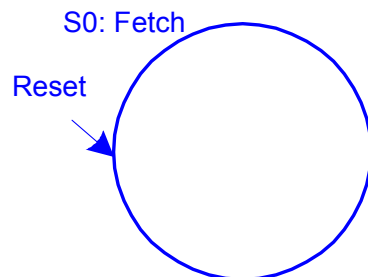
# Multicycle Processor



# Multicycle Control



# Main Controller FSM: Fetch

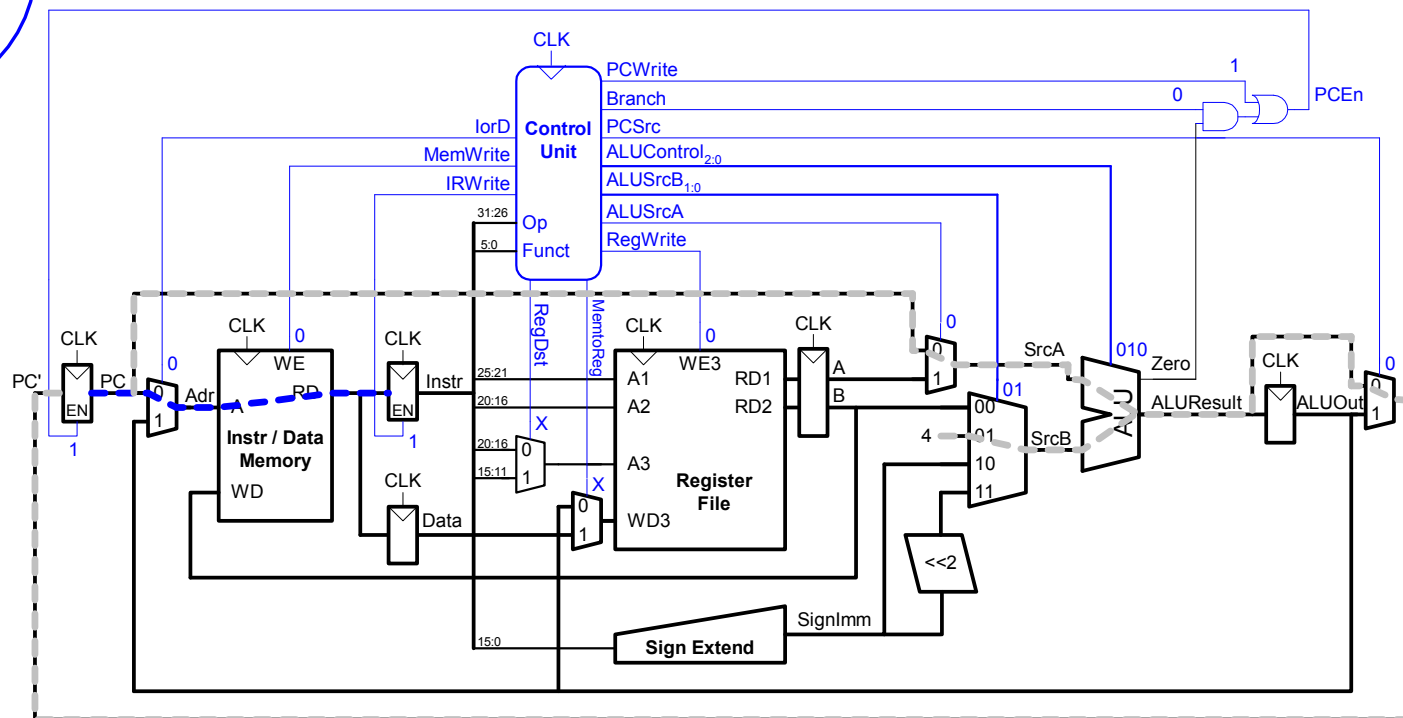


# Main Controller FSM: Fetch

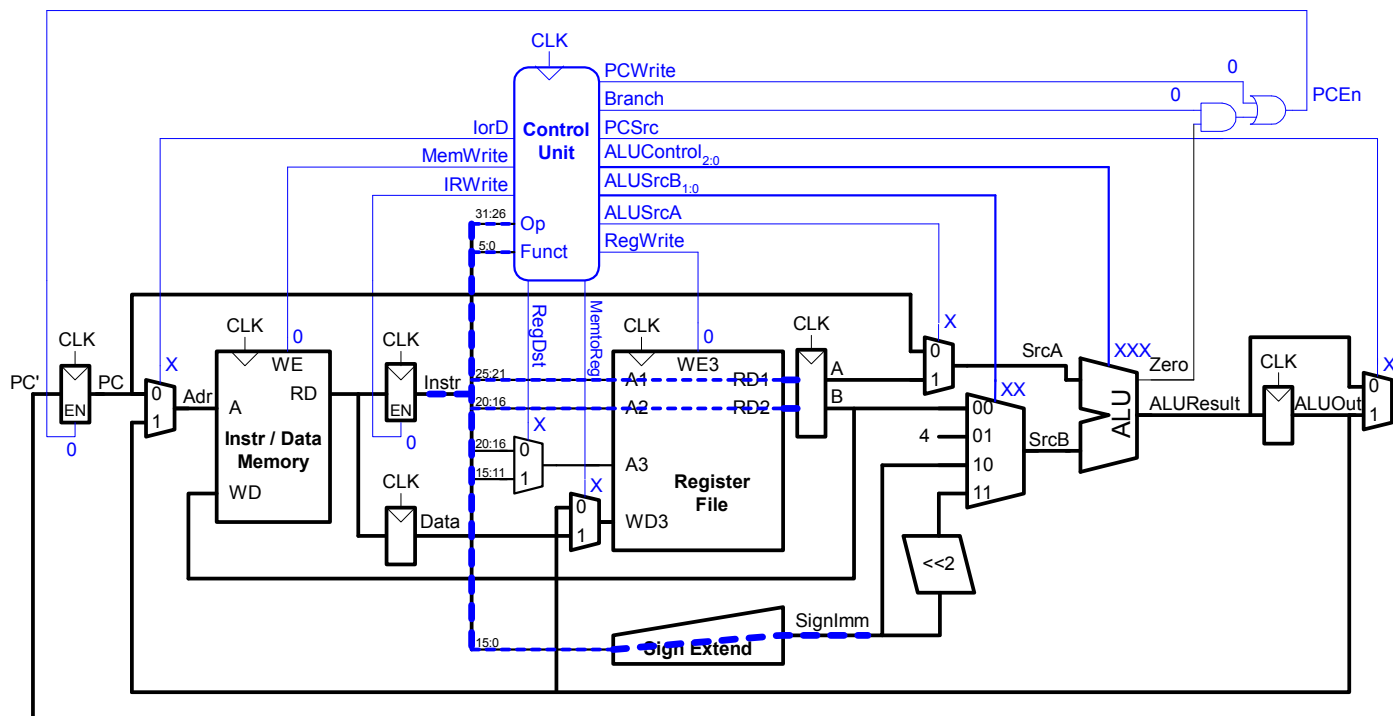
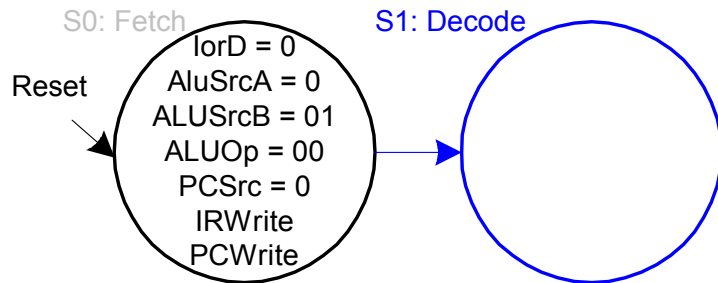
S0: Fetch

Reset

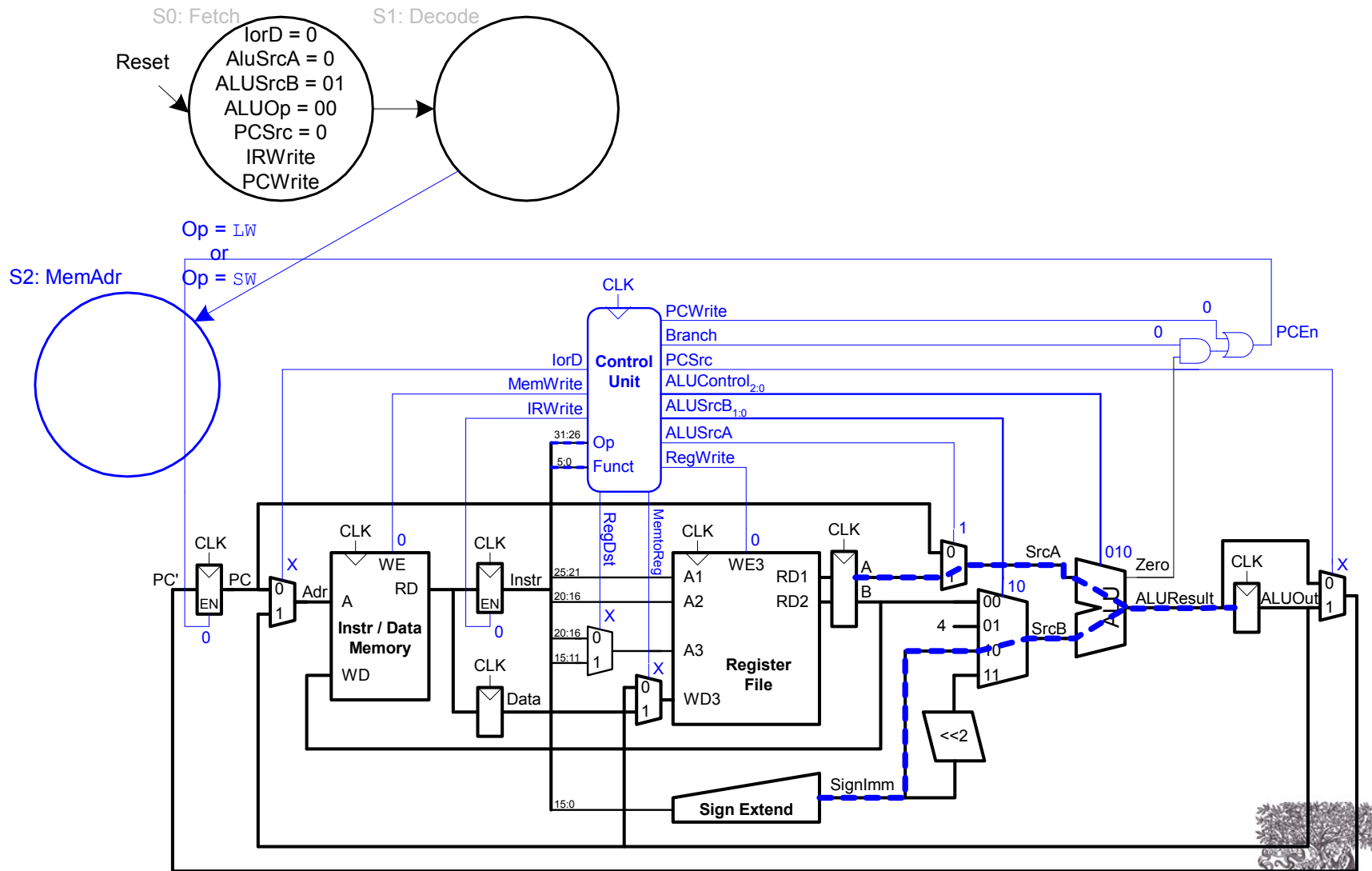
lrd = 0  
 AluSrcA = 0  
 ALUSrcB = 01  
 ALUOp = 00  
 PCSrc = 0  
 IRWrite  
 PCWrite



# Main Controller FSM: Decode

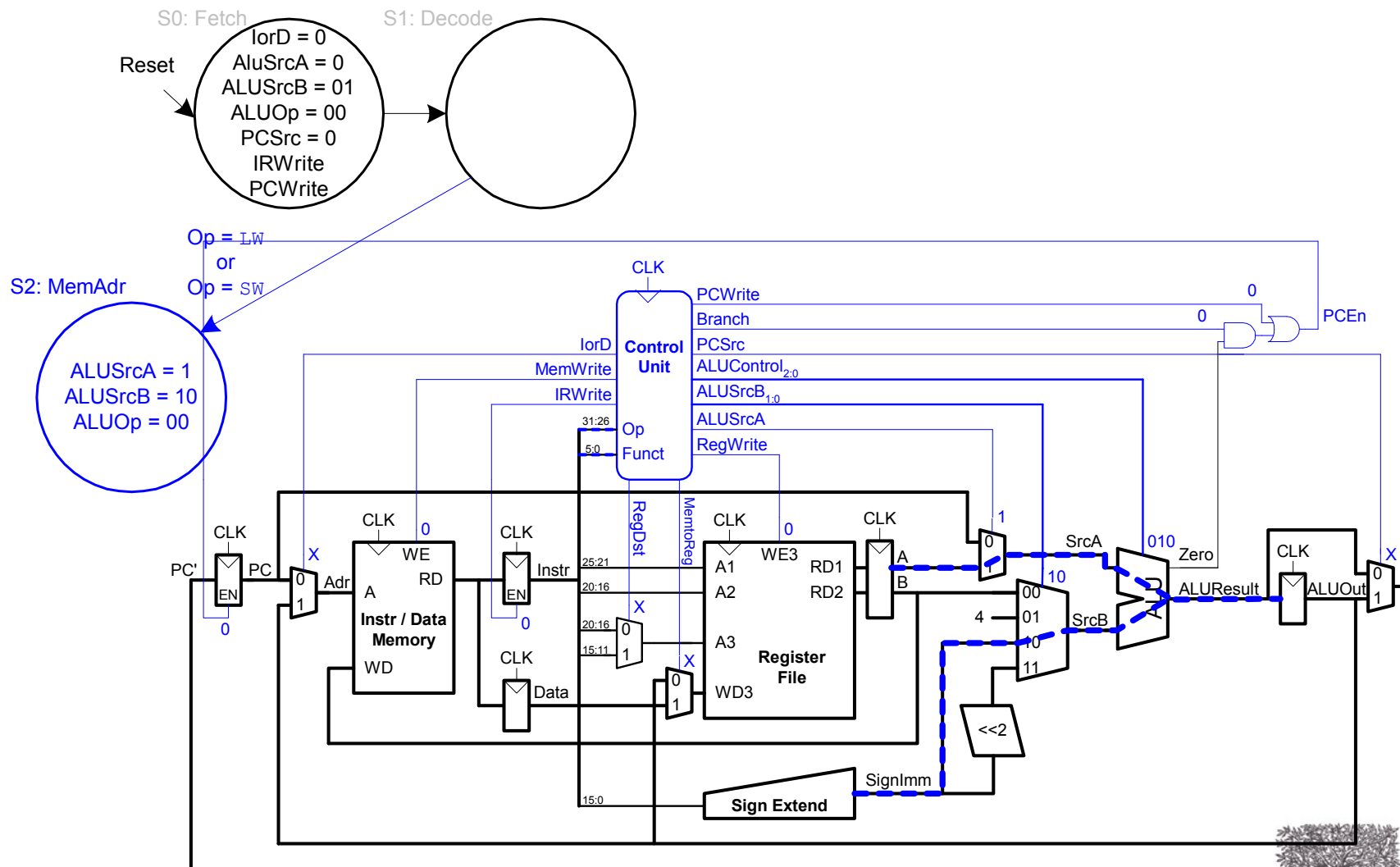


# Main Controller FSM: Address

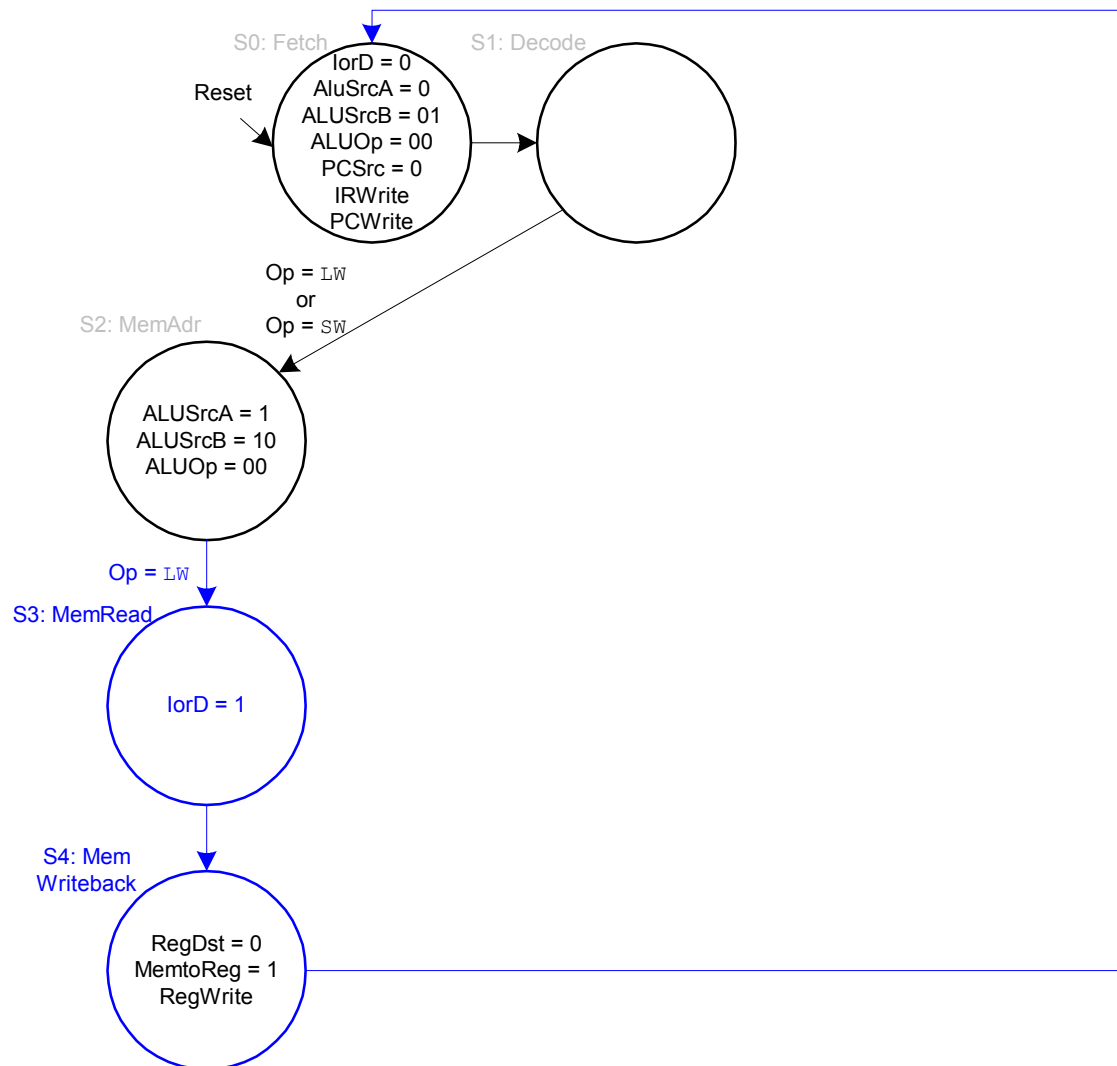




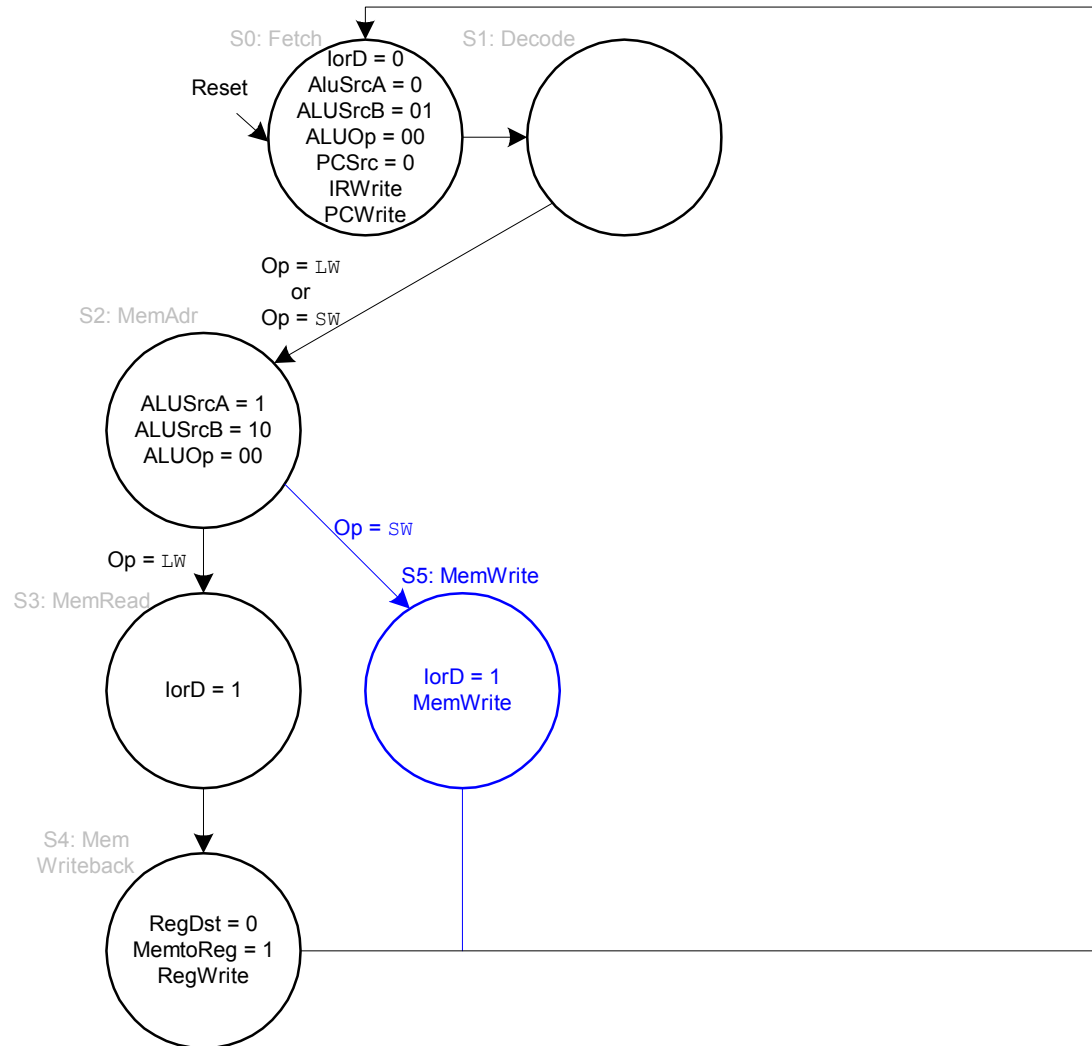
# Main Controller FSM: Address



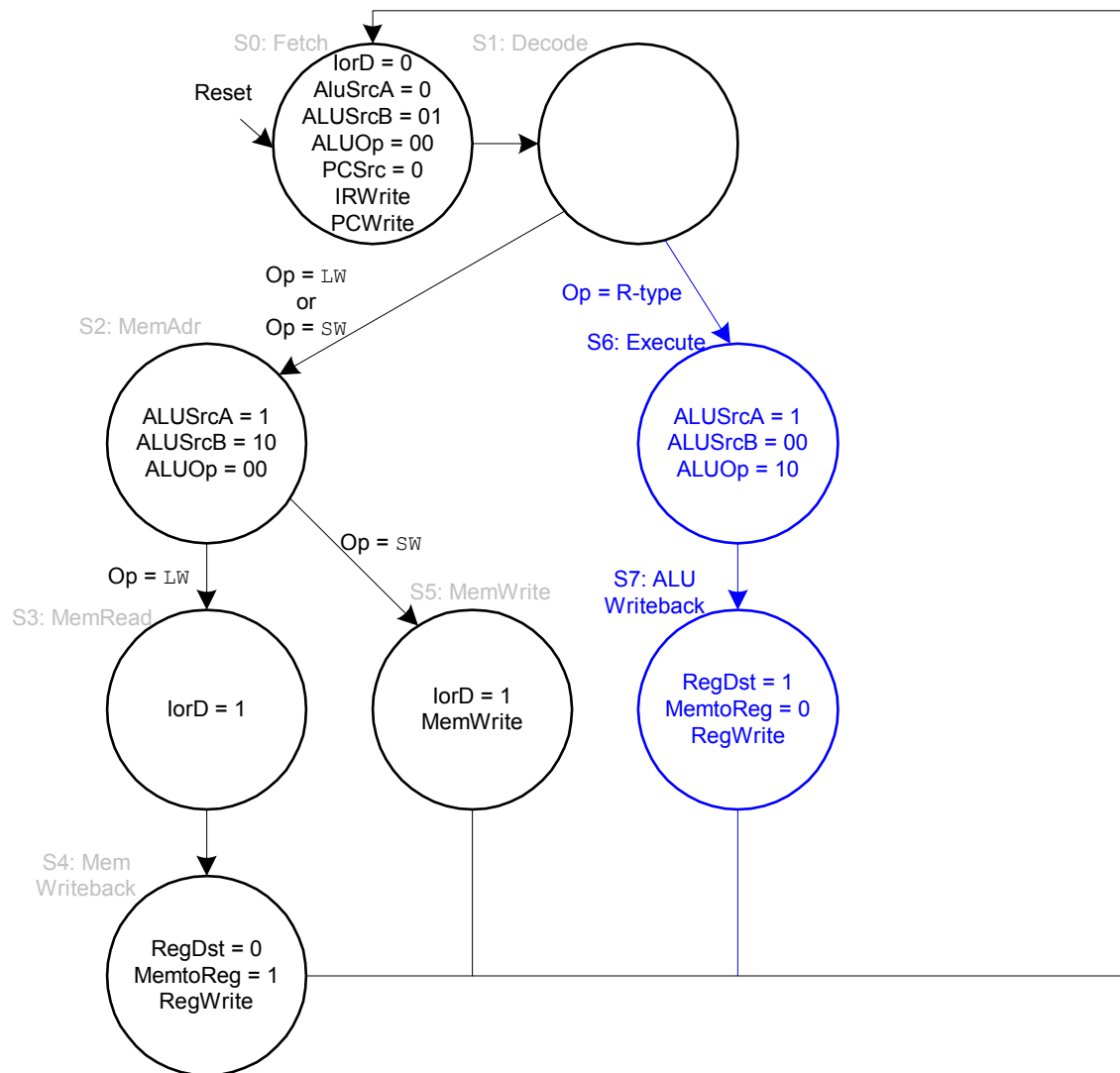
# Main Controller FSM: $\text{lw}$



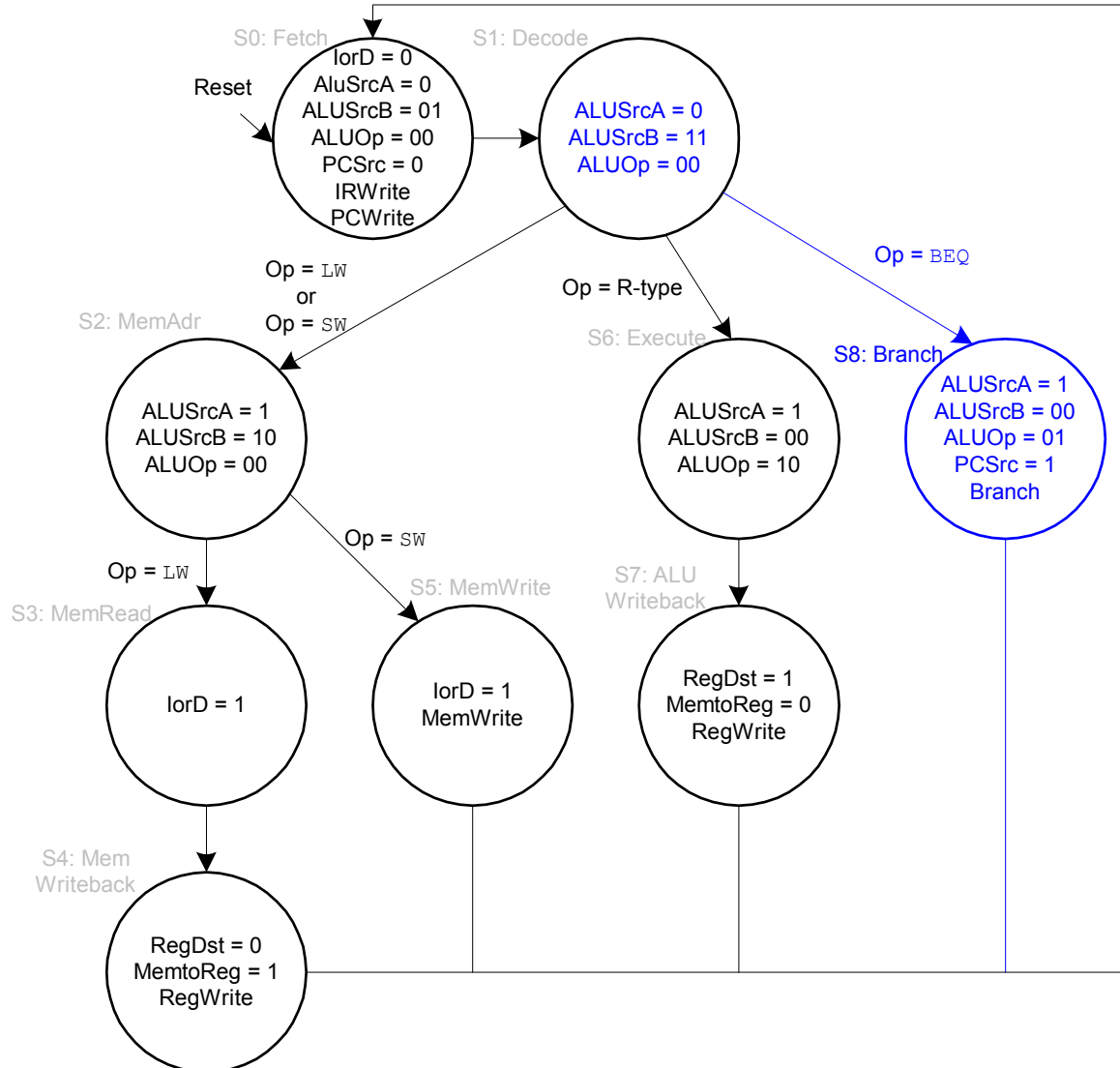
# Main Controller FSM: $sw$



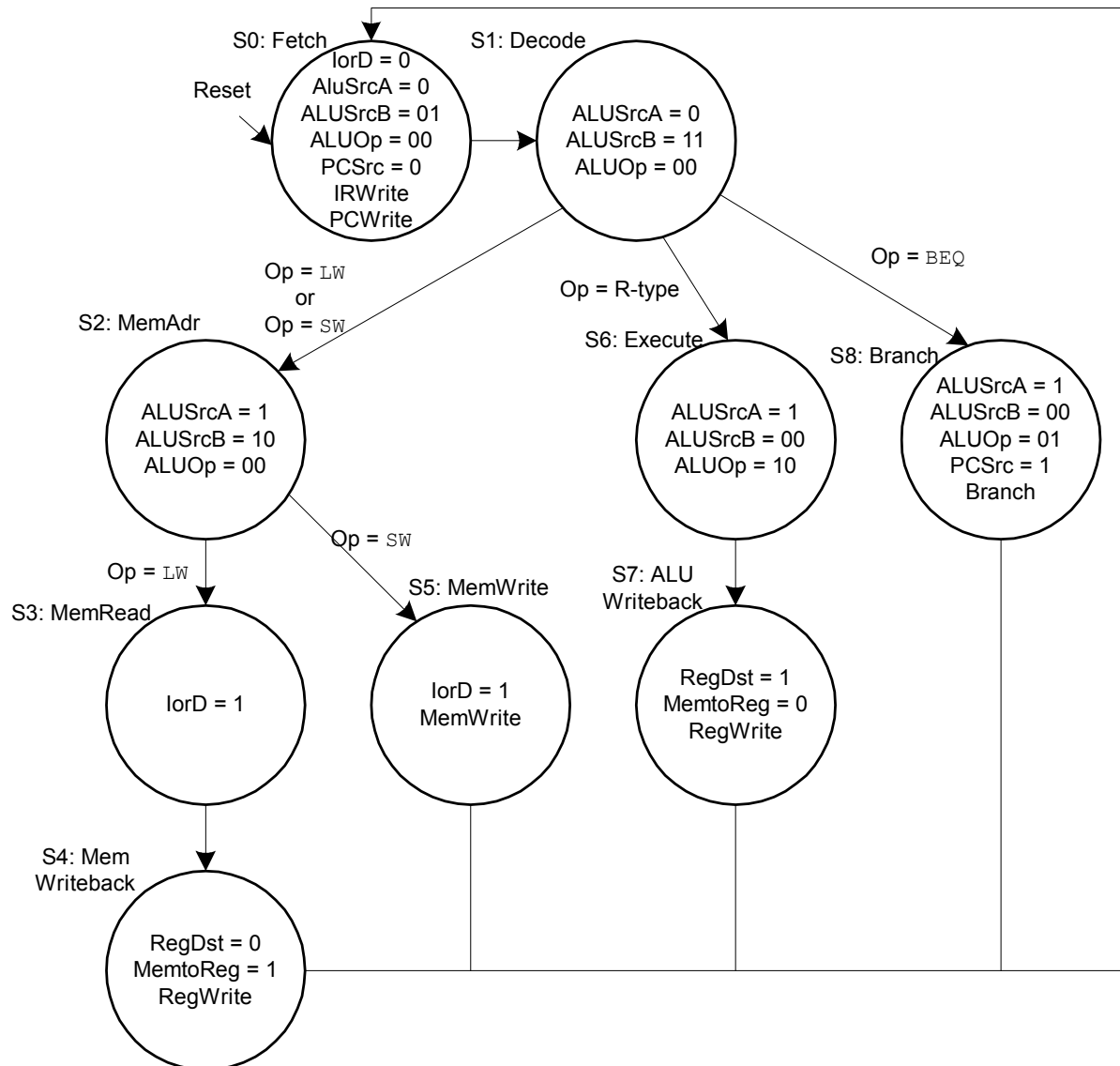
# Main Controller FSM: R-Type



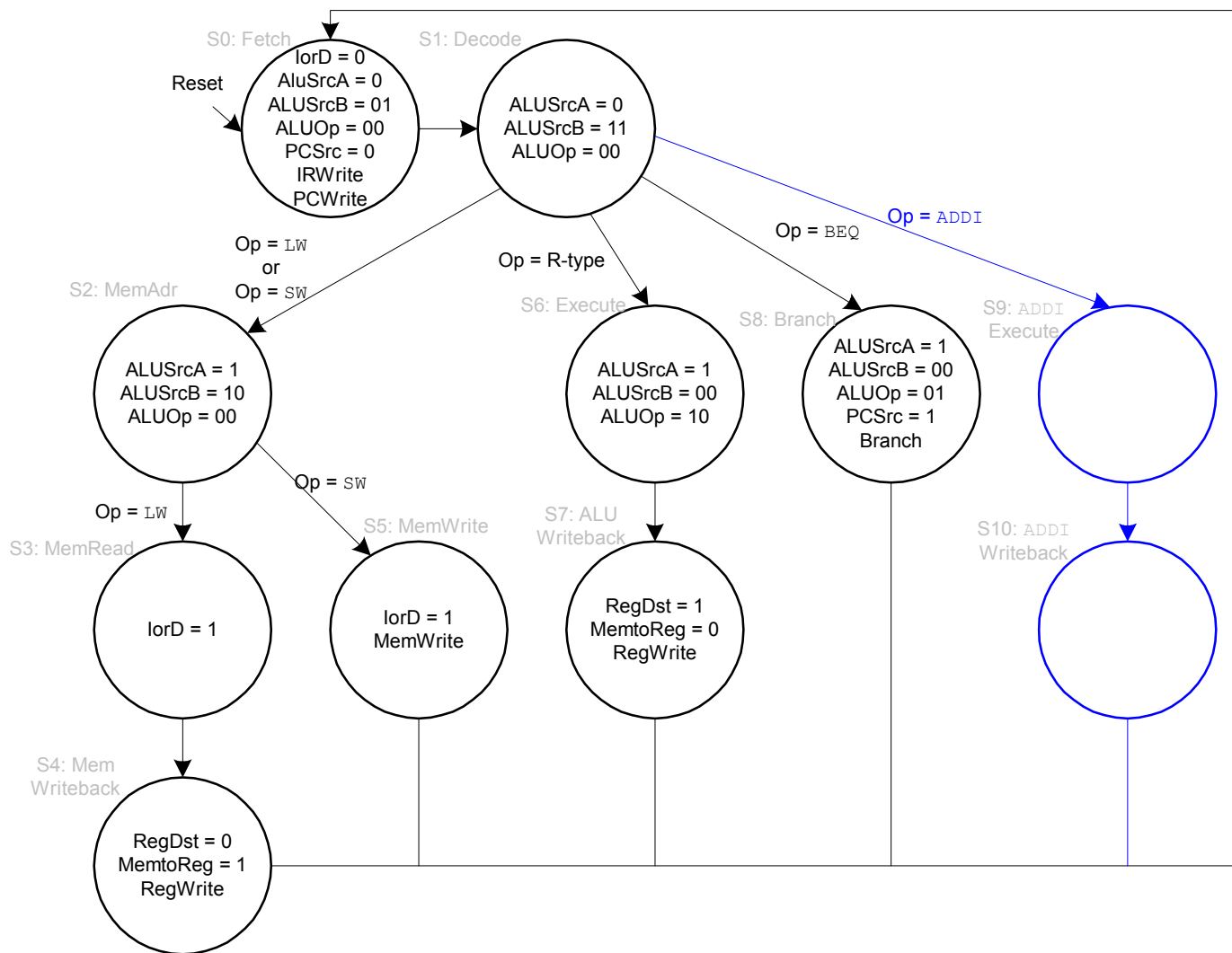
# Main Controller FSM: beq



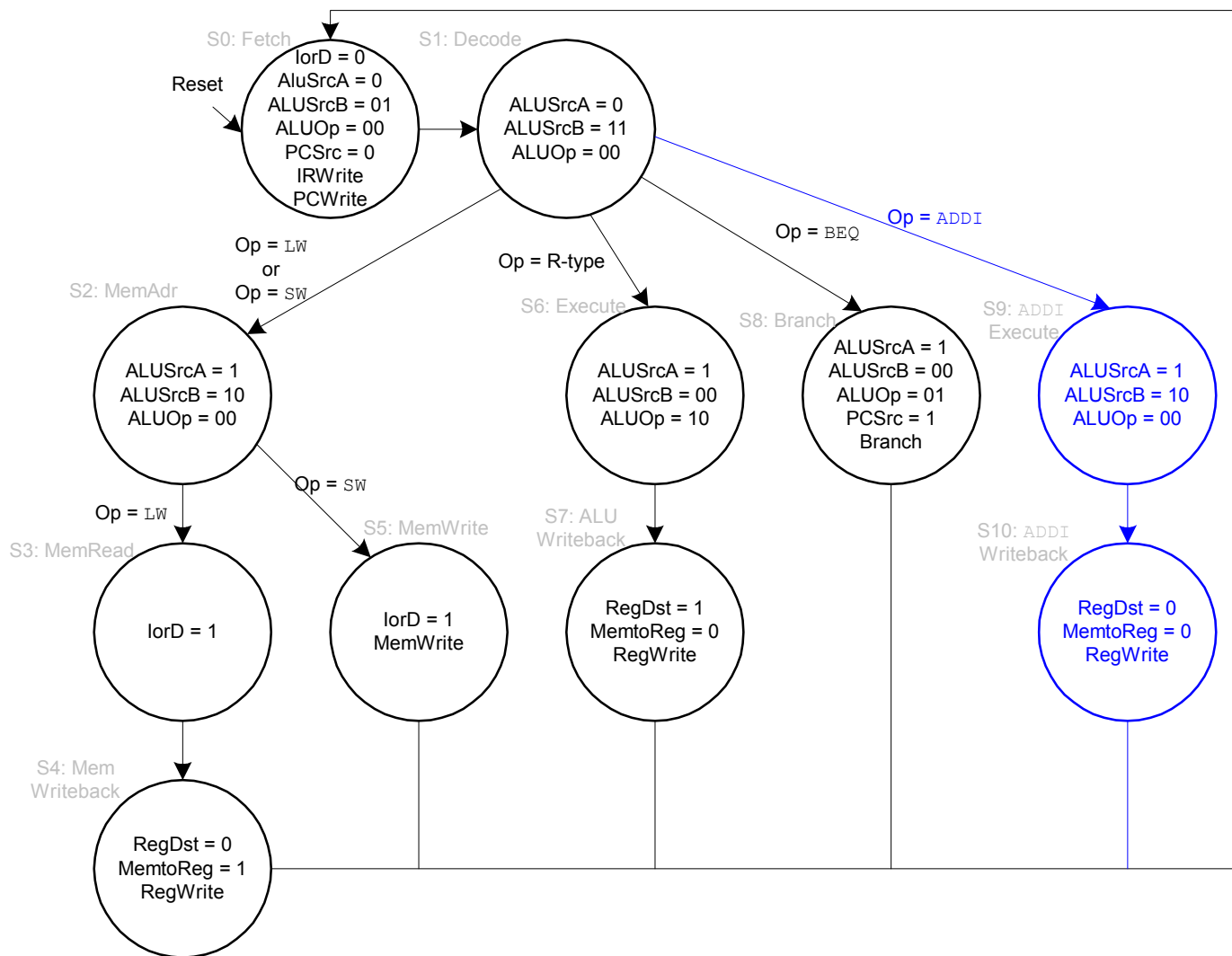
# Multicycle Controller FSM



# Extended Functionality: addi

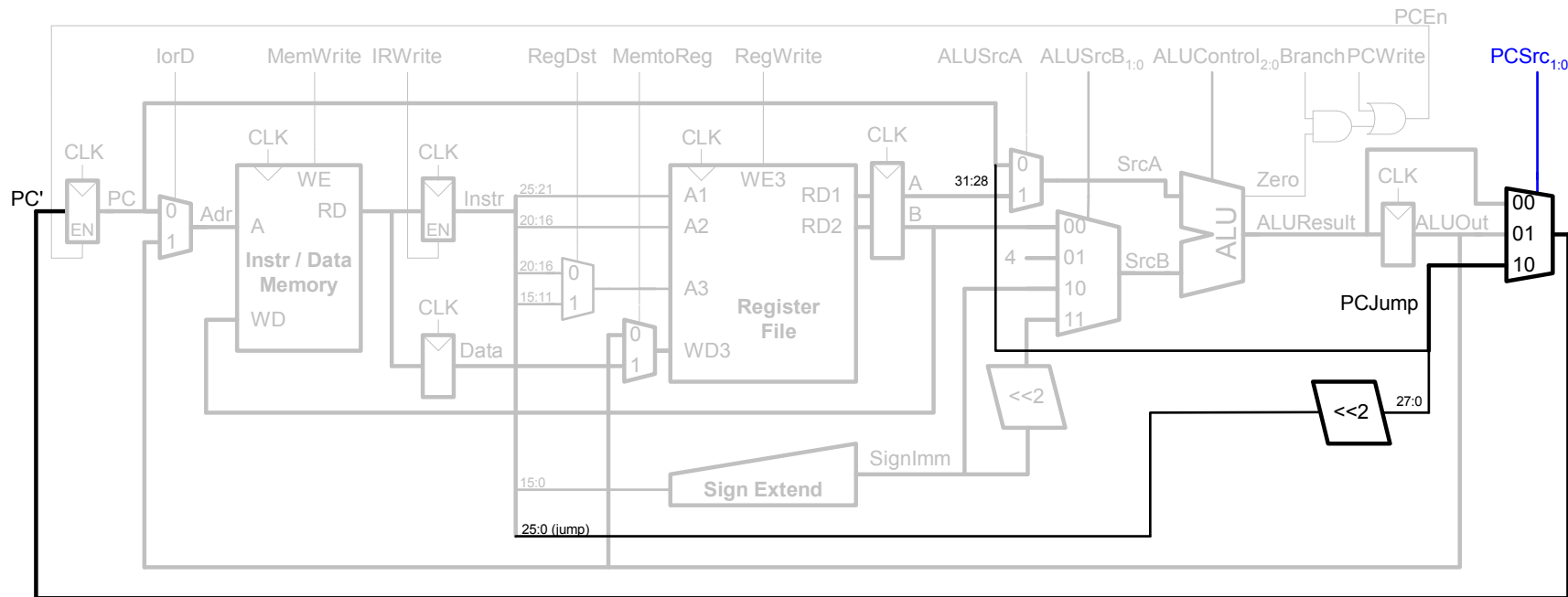


# Main Controller FSM: addi

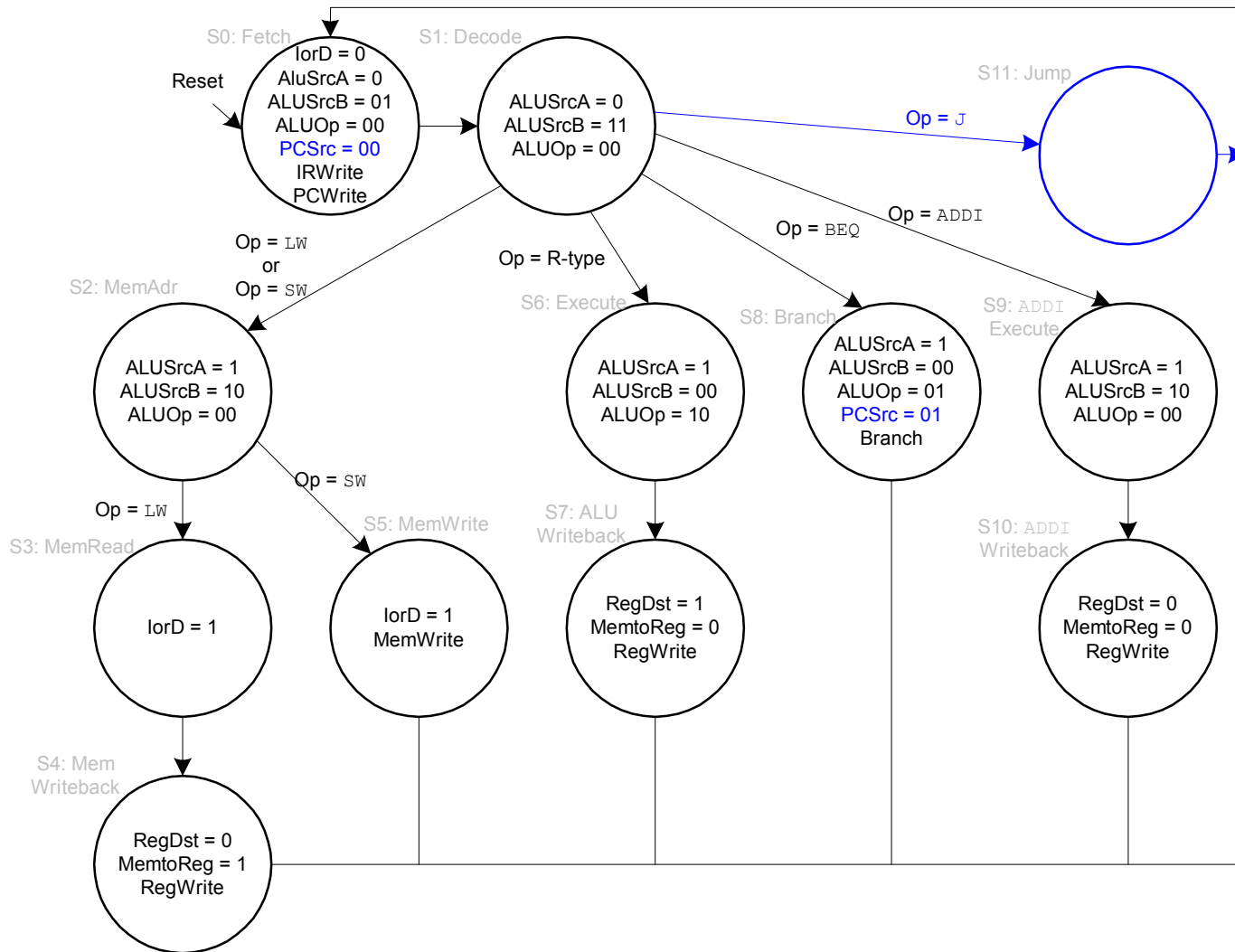




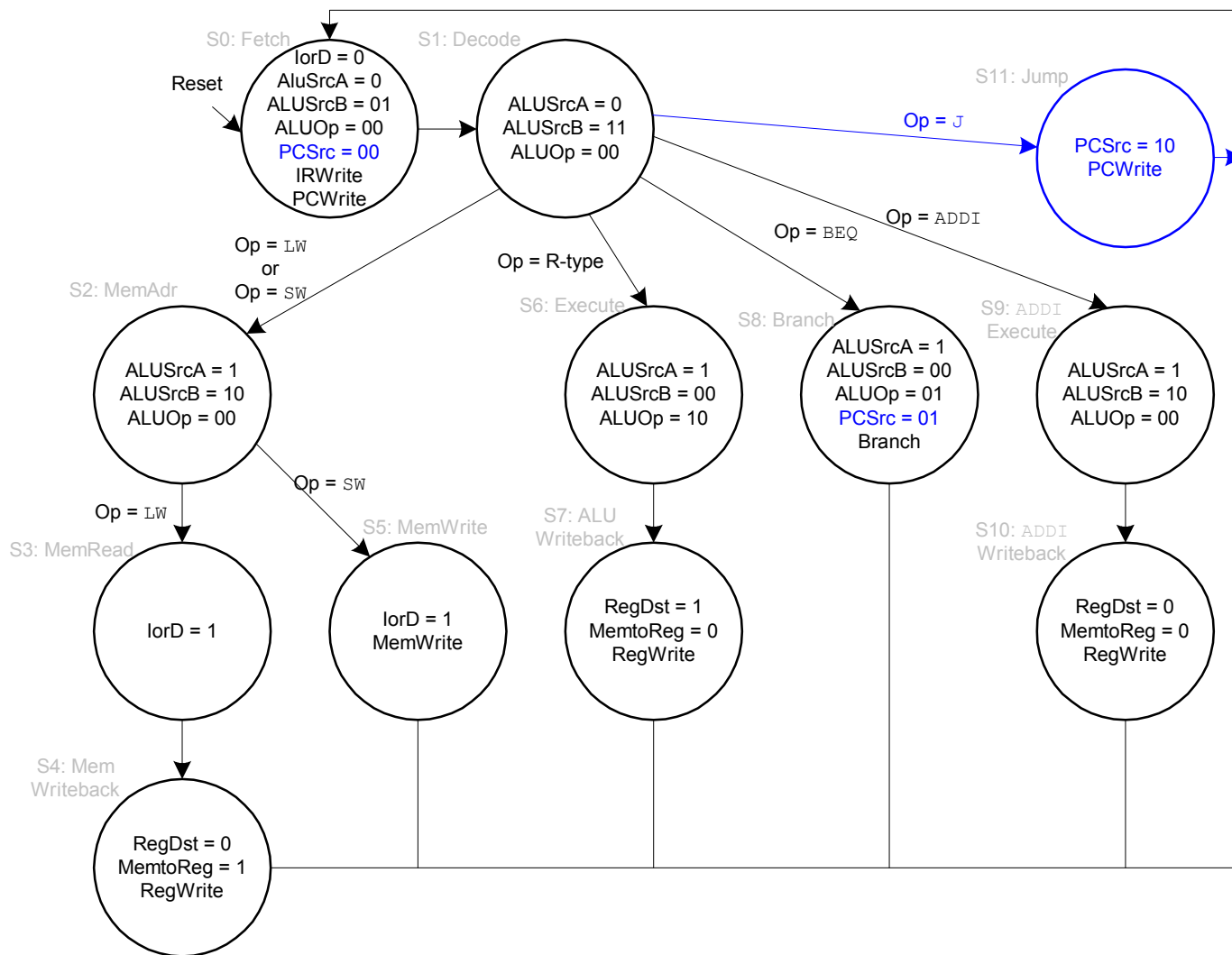
# Extended Functionality: j



# Main Controller FSM: j



# Main Controller FSM: j



# Multicycle Processor Performance

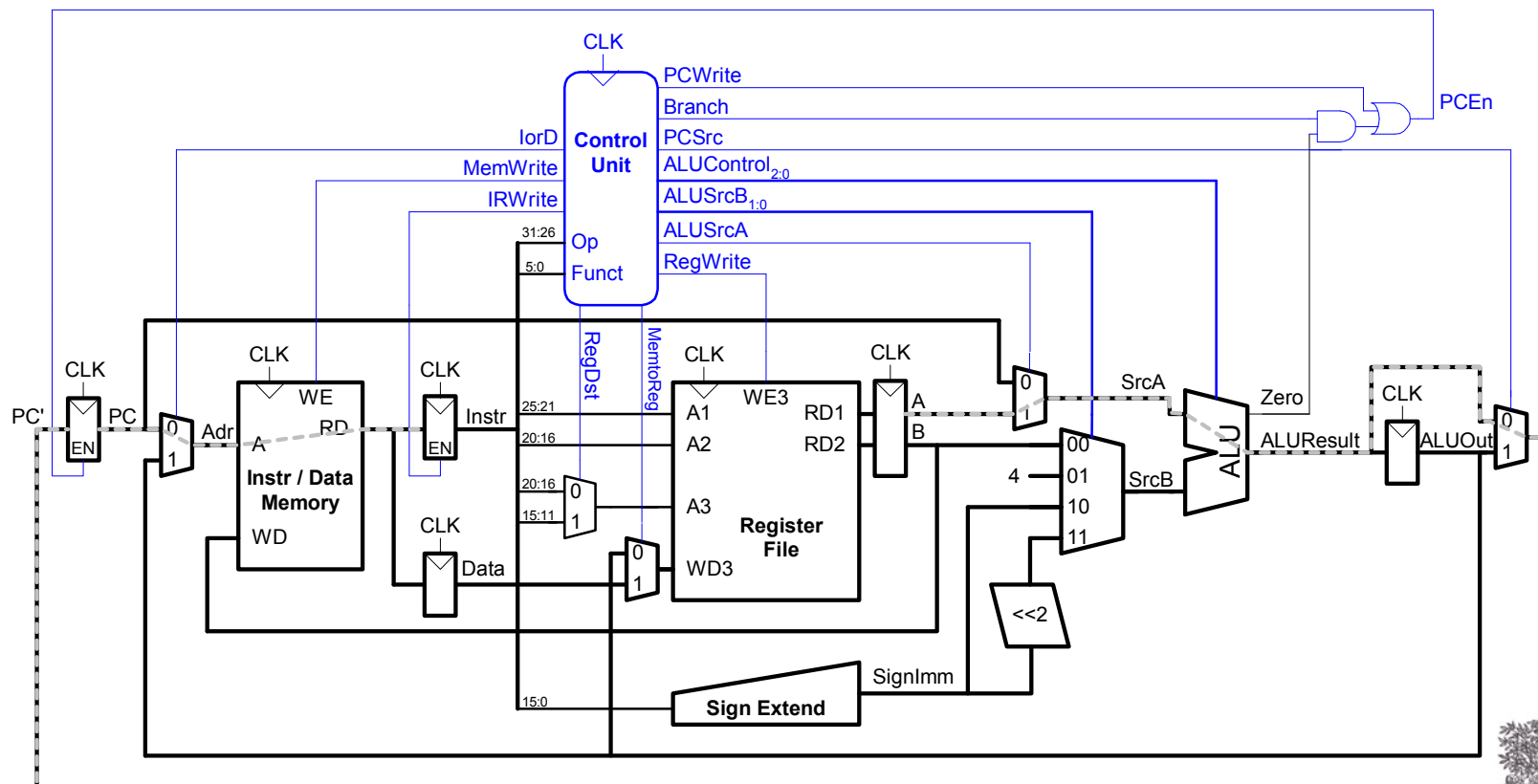
- Instructions take different number of cycles:
  - 3 cycles: beq, j
  - 4 cycles: R-Type, sw, addi
  - 5 cycles: lw
- CPI is weighted average
- SPECINT2000 benchmark:
  - 25% loads
  - 10% stores
  - 11% branches
  - 2% jumps
  - 52% R-type

**Average CPI** =  $(0.11 + 0.02)(3) + (0.52 + 0.10)(4) + (0.25)(5) = 4.12$

# Multicycle Processor Performance

Multicycle critical path:

$$T_c = t_{pcq} + t_{mux} + \max(t_{ALU} + t_{mux}, t_{mem}) + t_{setup}$$



# Multicycle Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	$t_{pcq\_PC}$	30
Register setup	$t_{setup}$	20
Multiplexer	$t_{mux}$	25
ALU	$t_{ALU}$	200
Memory read	$t_{mem}$	250
Register file read	$t_{RFread}$	150
Register file setup	$t_{RFsetup}$	20

$$T_c = ?$$

# Multicycle Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	$t_{pcq\_PC}$	30
Register setup	$t_{setup}$	20
Multiplexer	$t_{mux}$	25
ALU	$t_{ALU}$	200
Memory read	$t_{mem}$	250
Register file read	$t_{RFread}$	150
Register file setup	$t_{RFsetup}$	20

$$\begin{aligned}
 T_c &= t_{pcq\_PC} + t_{mux} + \max(t_{ALU} + t_{mux}, t_{mem}) + t_{setup} \\
 &= t_{pcq\_PC} + t_{mux} + t_{mem} + t_{setup} \\
 &= [30 + 25 + 250 + 20] \text{ ps} \\
 &= \mathbf{325 \text{ ps}}
 \end{aligned}$$

# Multicycle Performance Example

Program with 100 billion instructions

Execution Time = ?



# Multicycle Performance Example

Program with 100 billion instructions

$$\begin{aligned}\text{Execution Time} &= (\# \text{ instructions}) \times \text{CPI} \times T_c \\ &= (100 \times 10^9)(4.12)(325 \times 10^{-12}) \\ &= \mathbf{133.9 \text{ seconds}}\end{aligned}$$

This is **slower** than the single-cycle processor (92.5 seconds). Why?

# Multicycle Performance Example

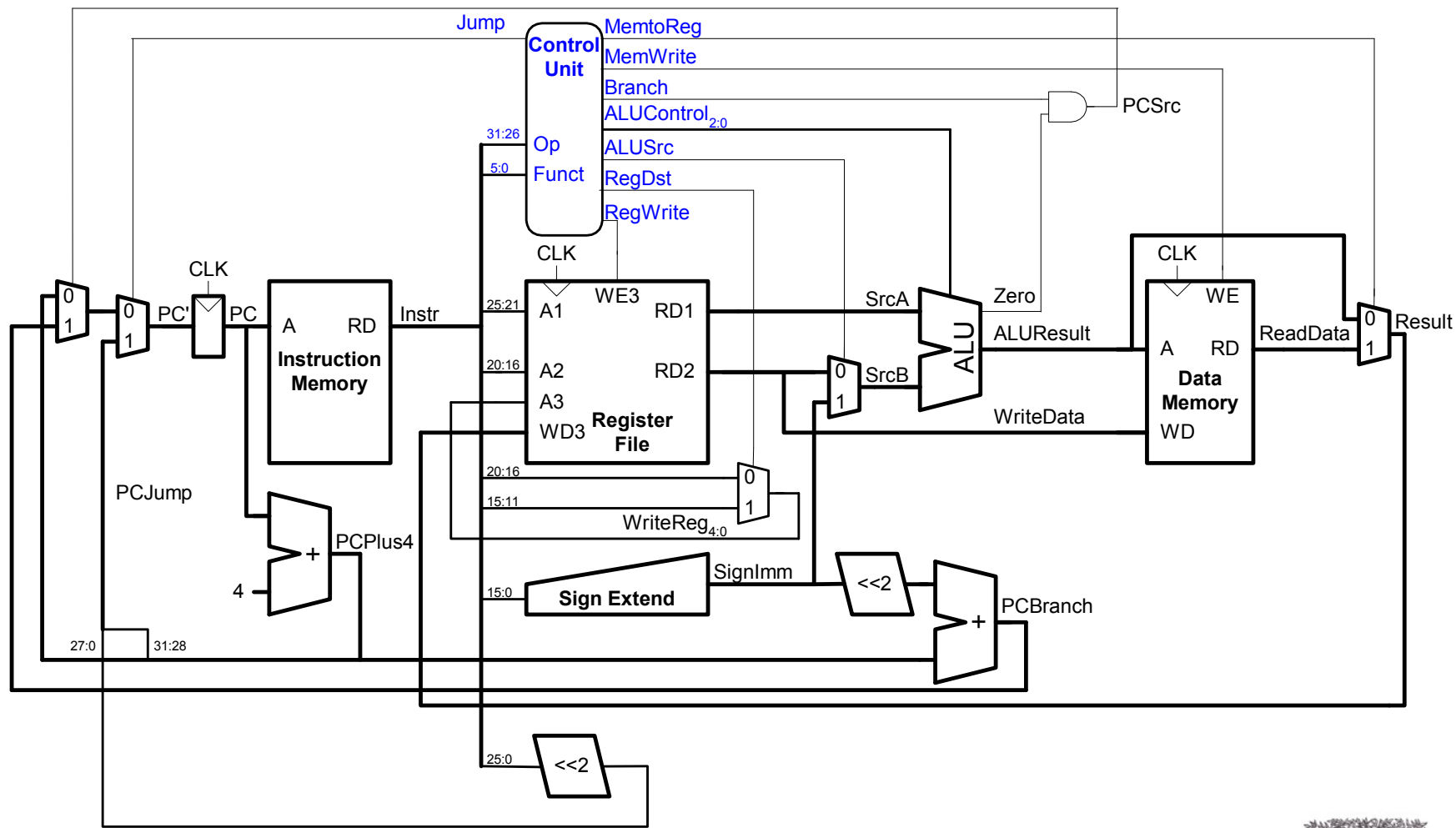
Program with 100 billion instructions

$$\begin{aligned}\text{Execution Time} &= (\# \text{ instructions}) \times \text{CPI} \times T_c \\ &= (100 \times 10^9)(4.12)(325 \times 10^{-12}) \\ &= \mathbf{133.9 \text{ seconds}}\end{aligned}$$

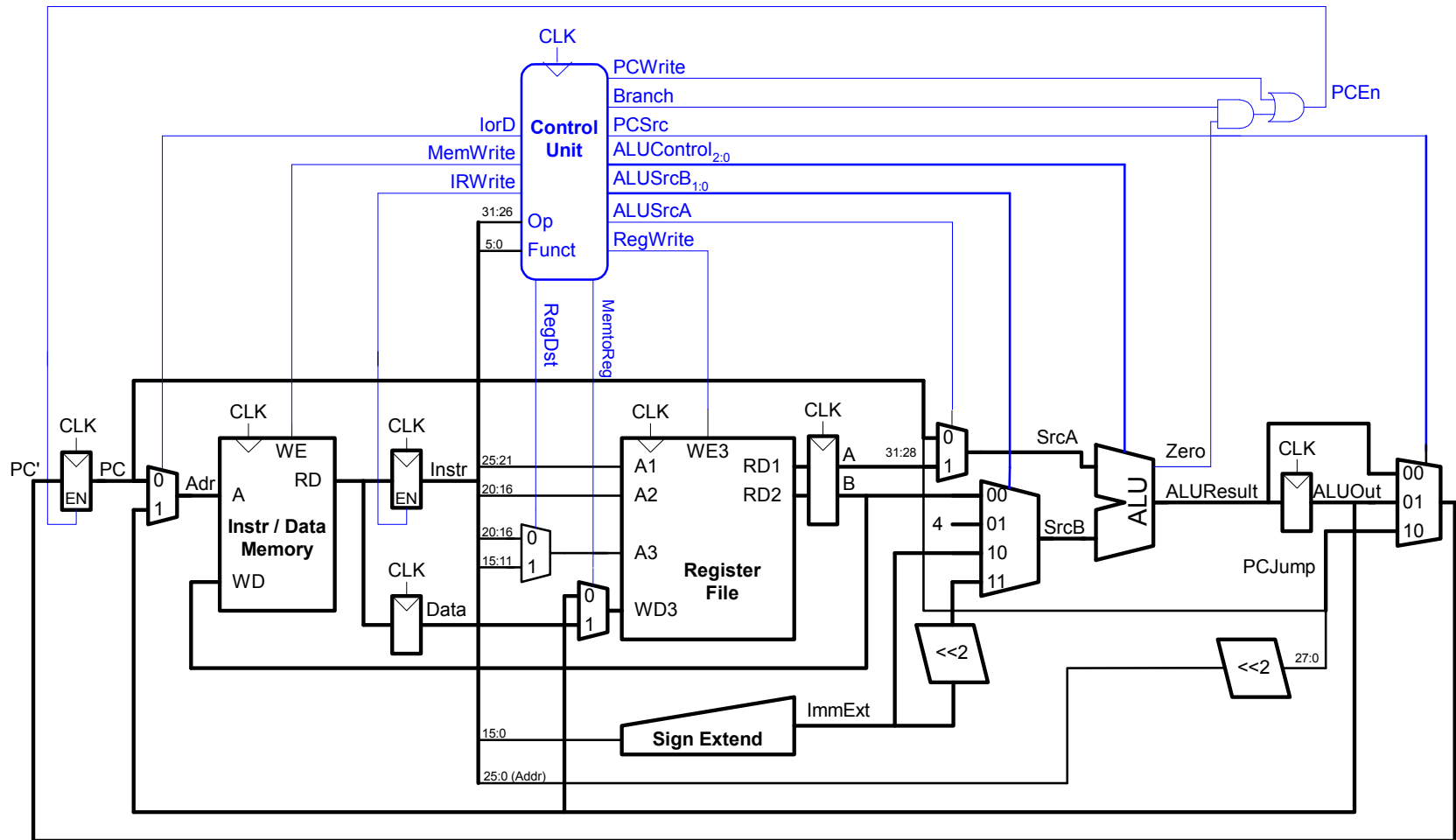
This is **slower** than the single-cycle processor (92.5 seconds). Why?

- Not all steps same length
- Sequencing overhead for each step ( $t_{pcq} + t_{\text{setup}} = 50 \text{ ps}$ )

# Review: Single-Cycle Processor



# Review: Multicycle Processor

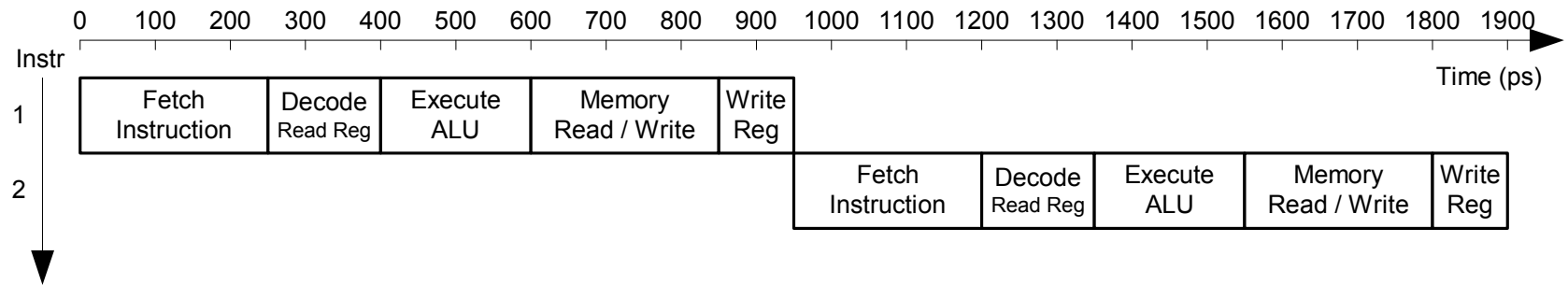


# Pipelined MIPS Processor

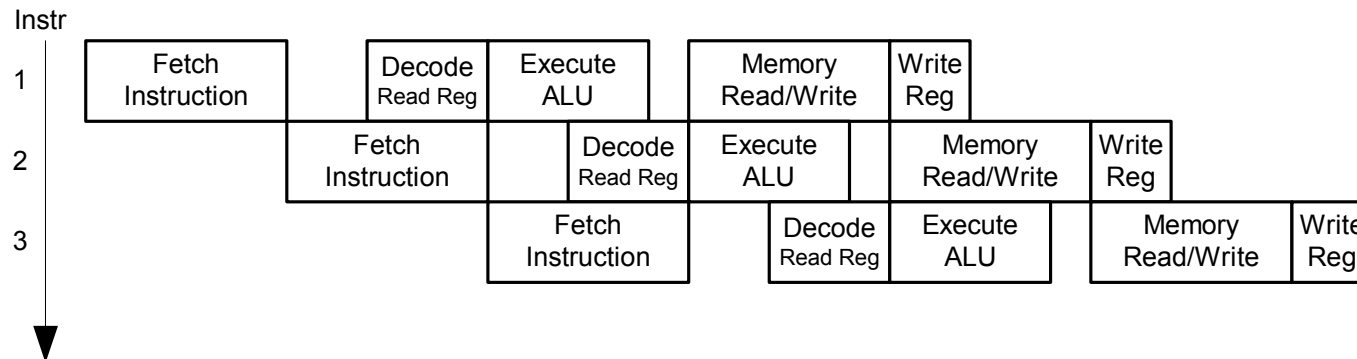
- Temporal parallelism
- Divide single-cycle processor into 5 stages:
  - Fetch
  - Decode
  - Execute
  - Memory
  - Writeback
- Add pipeline registers between stages

# Single-Cycle vs. Pipelined

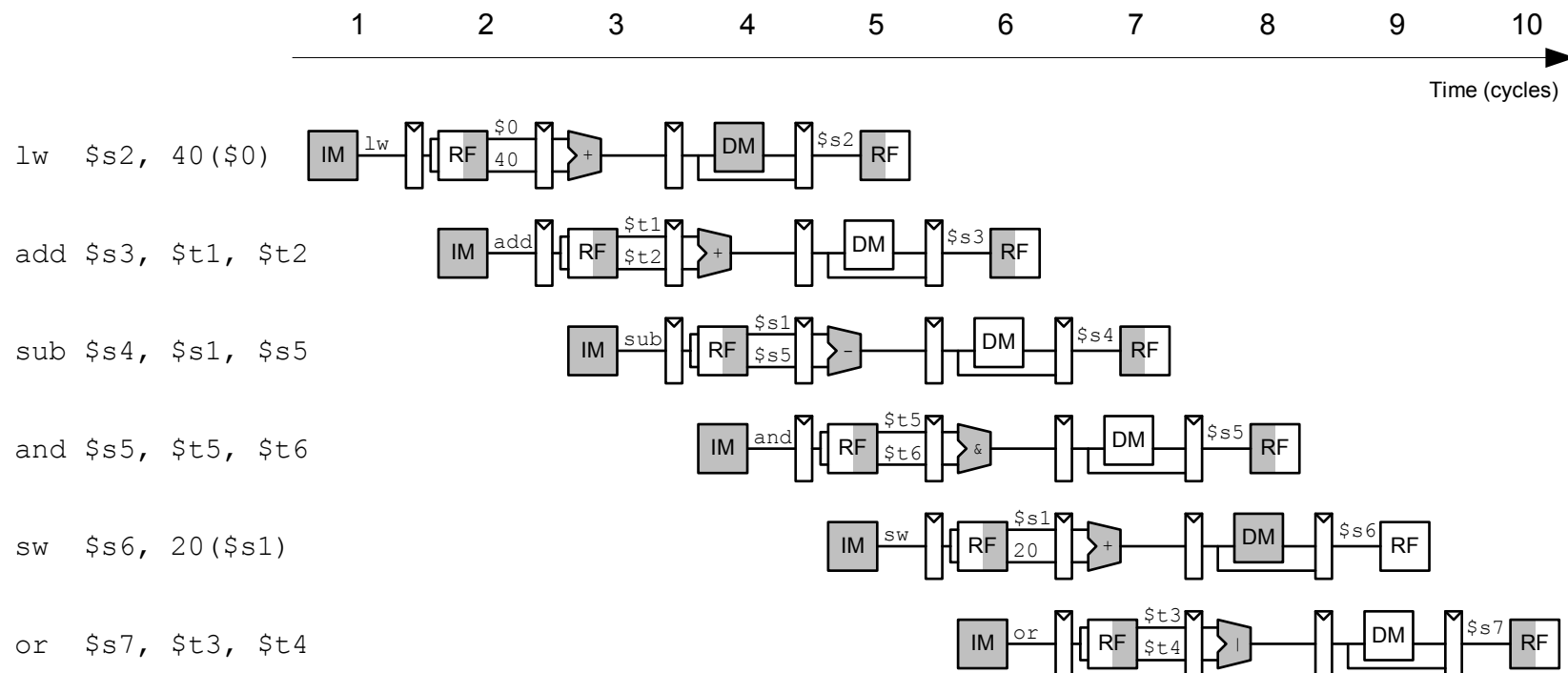
## Single-Cycle



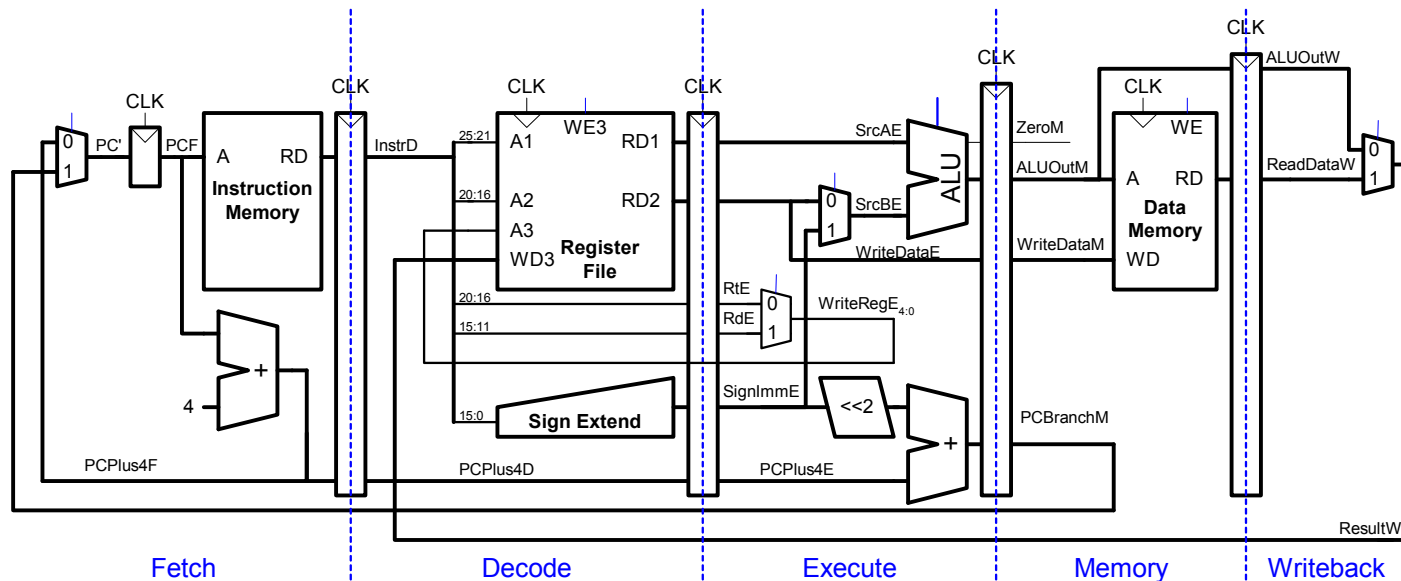
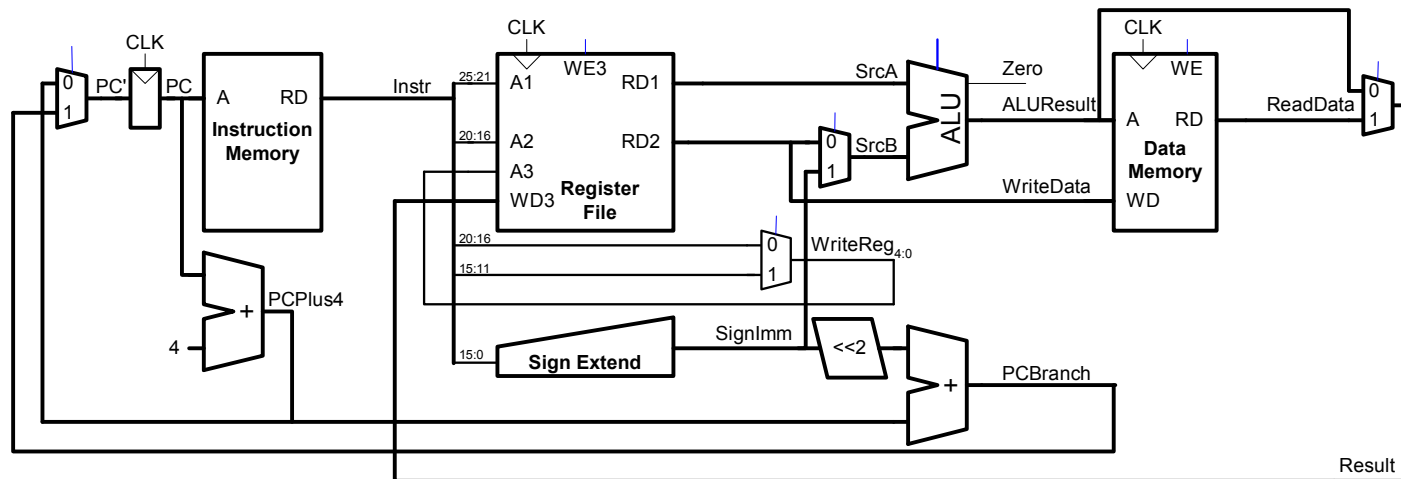
## Pipelined



# Pipelined Processor Abstraction

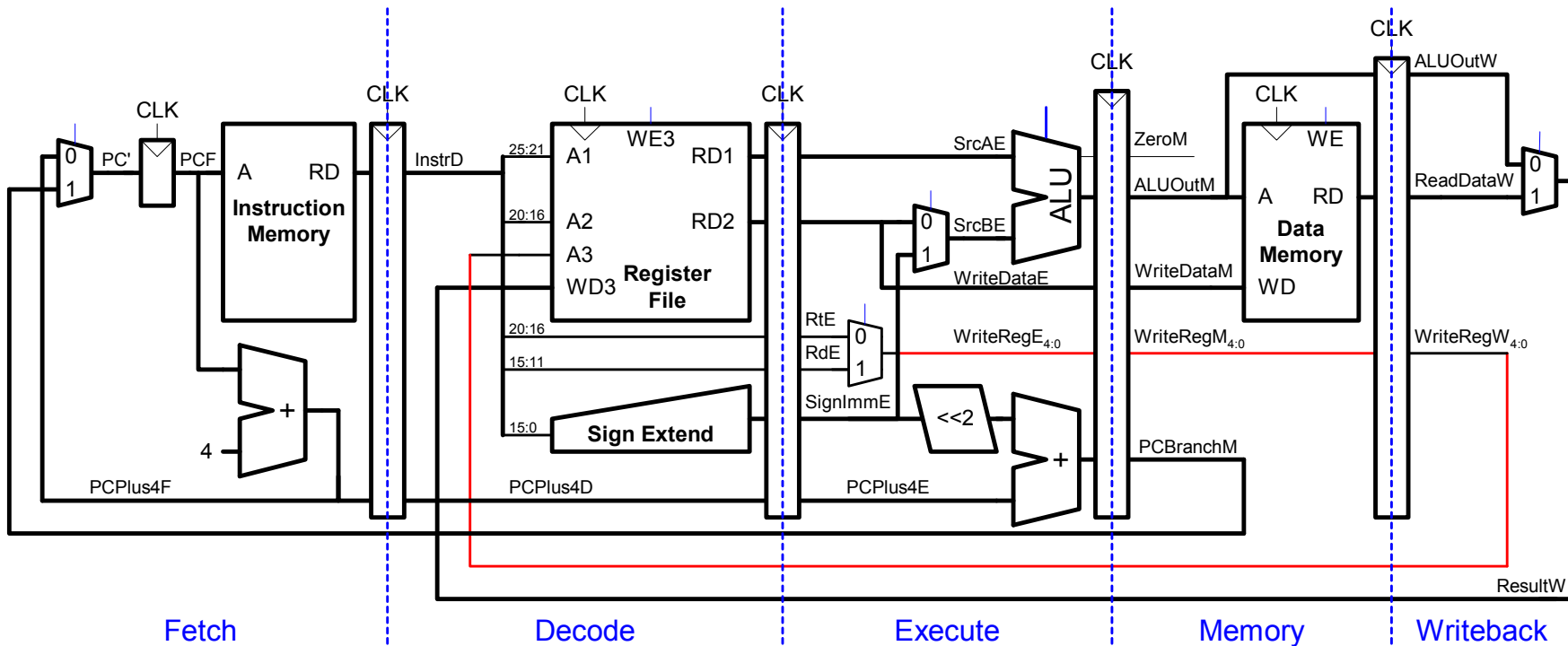


# Single-Cycle & Pipelined Datapath



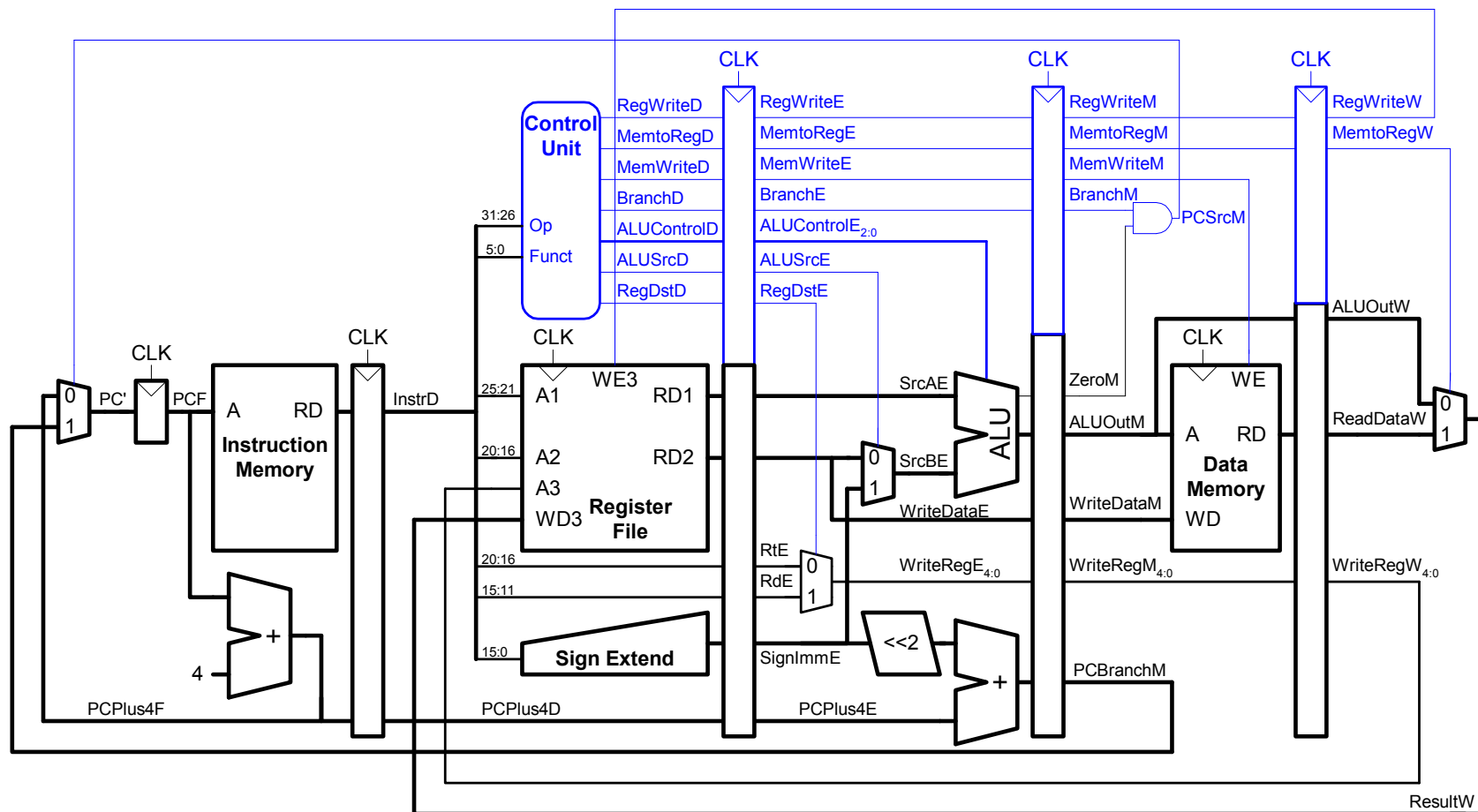


# Corrected Pipelined Datapath



*WriteReg must arrive at same time as Result*

# Pipelined Processor Control

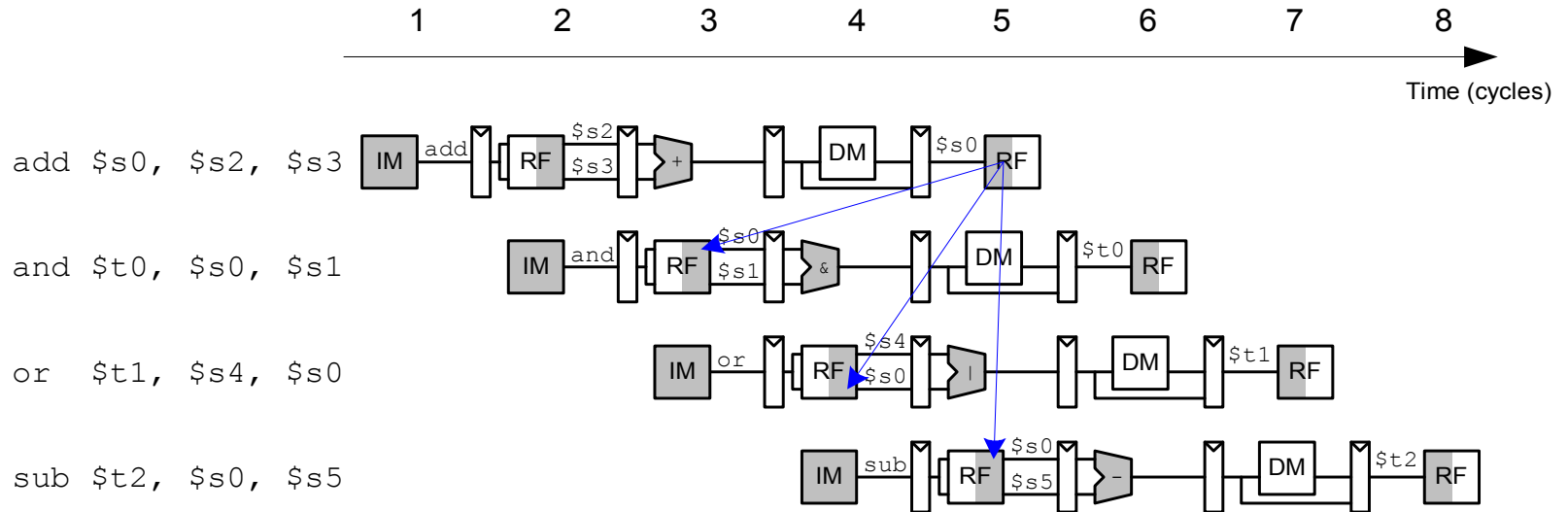


- Same control unit as single-cycle processor
- Control delayed to proper pipeline stage

# Pipeline Hazards

- When an instruction depends on result from instruction that hasn't completed
- Types:
  - **Data hazard:** register value not yet written back to register file
  - **Control hazard:** next instruction not decided yet (caused by branches)

# Data Hazard

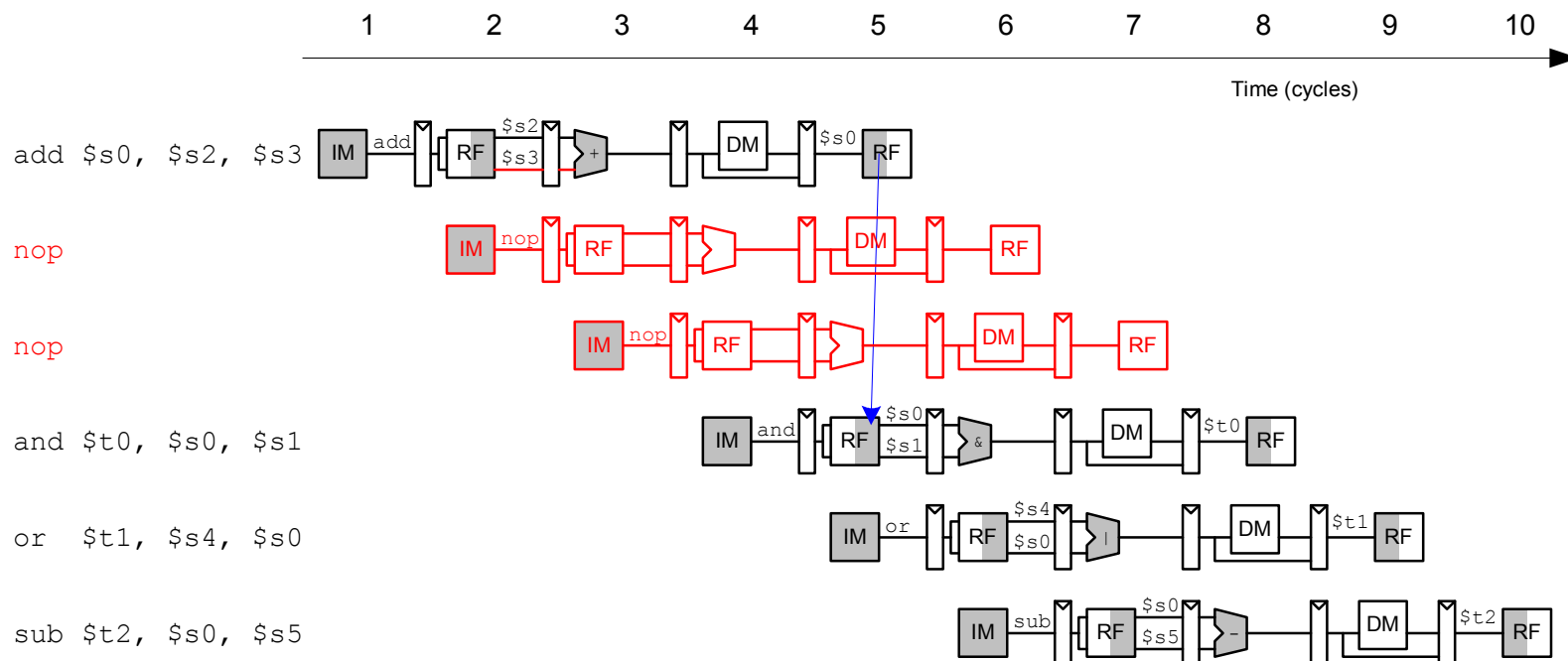


# Handling Data Hazards

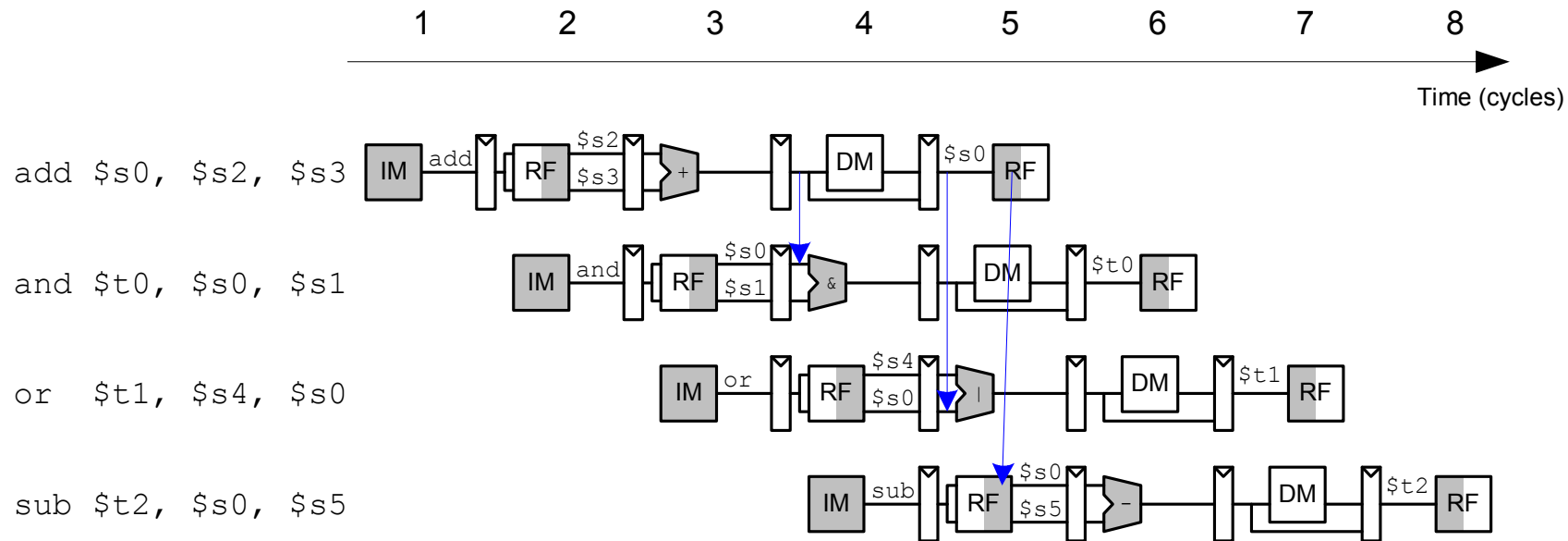
- Insert `nops` in code at compile time
- Rearrange code at compile time
- Forward data at run time
- Stall the processor at run time

# Compile-Time Hazard Elimination

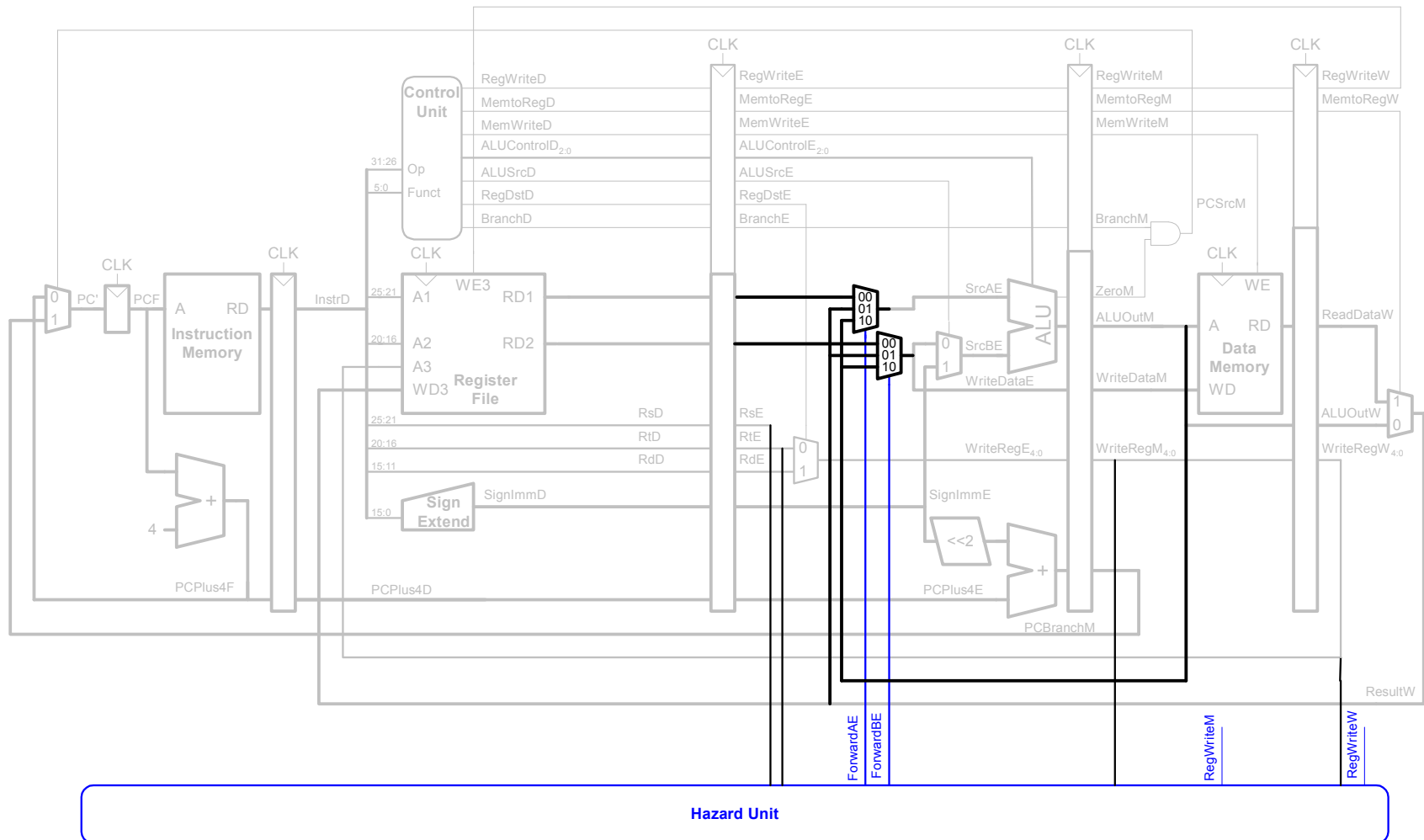
- Insert enough nops for result to be ready
- Or move independent useful instructions forward



# Data Forwarding



# Data Forwarding





# Data Forwarding

- Forward to Execute stage from either:
  - Memory stage or
  - Writeback stage
- Forwarding logic for *ForwardAE*:

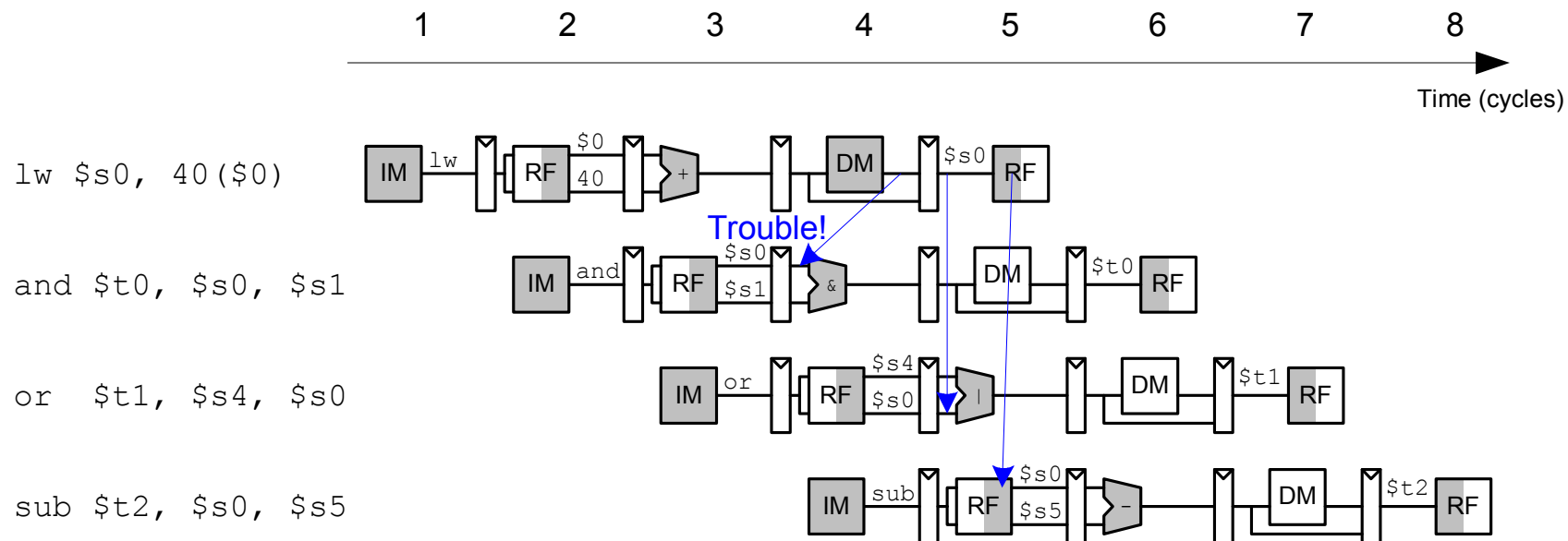
```
if      ((rsE != 0) AND (rsE == WriteRegM) AND RegWriteM)

    then  ForwardAE = 10
else if ((rsE != 0) AND (rsE == WriteRegW) AND RegWriteW)
    then  ForwardAE = 01
else      ForwardAE = 00
```

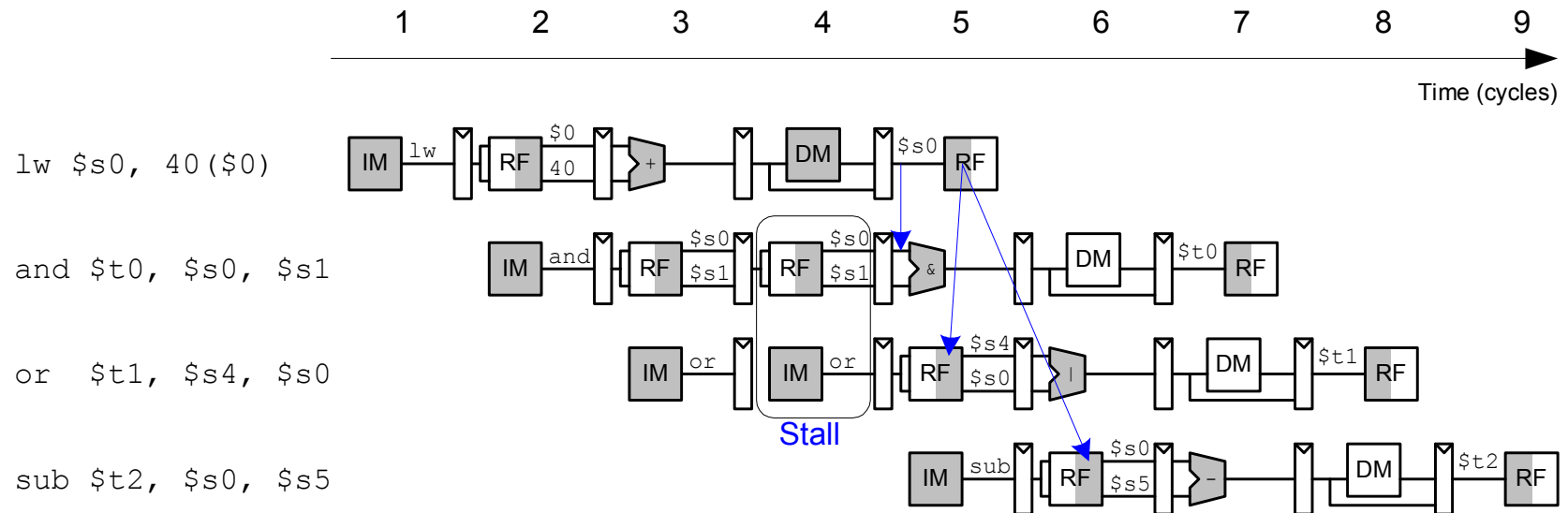
Forwarding logic for *ForwardBE* same, but replace *rsE* with *rtE*



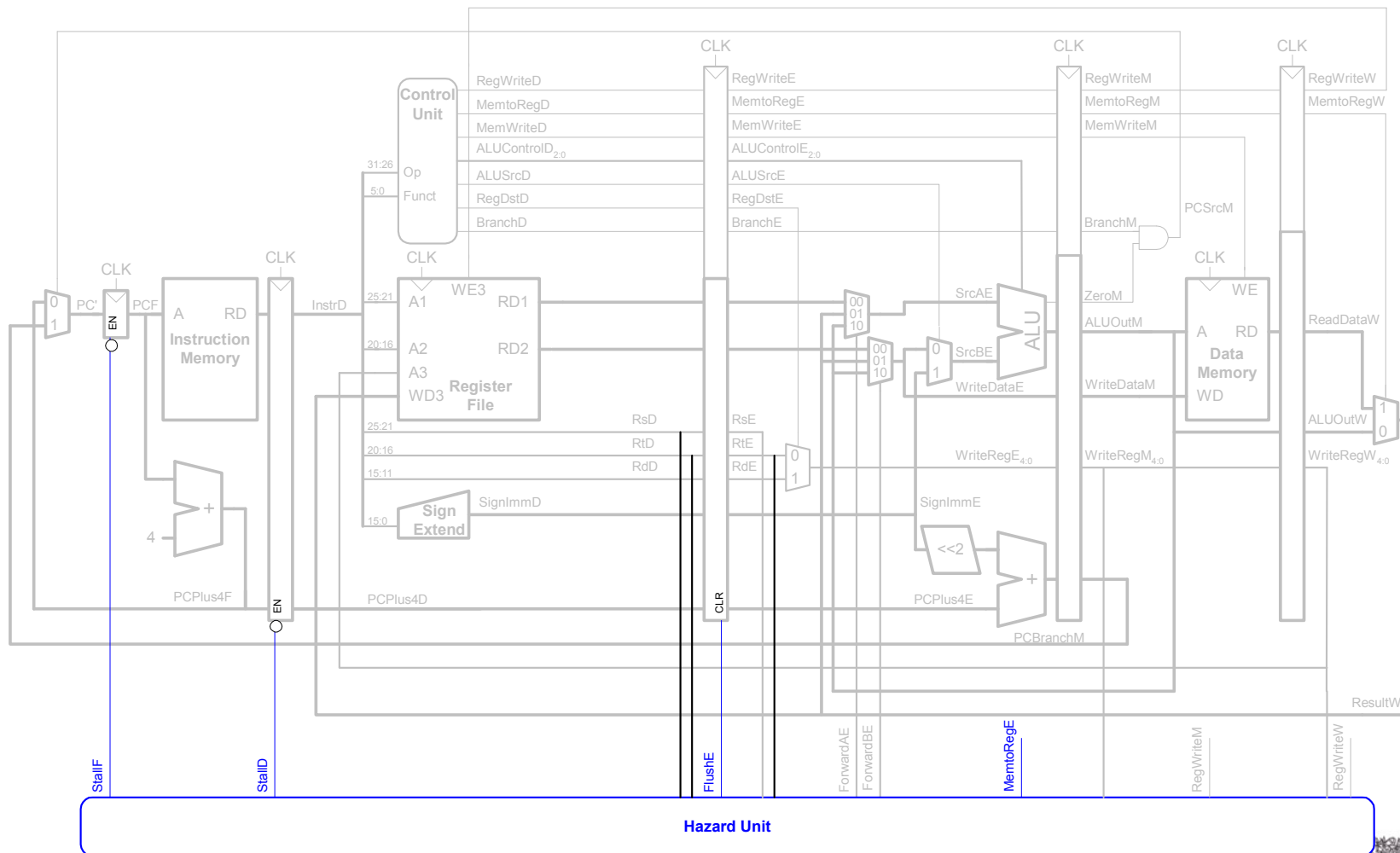
# Stalling



# Stalling



# Stalling Hardware



# Stalling Logic

*lwstall* =

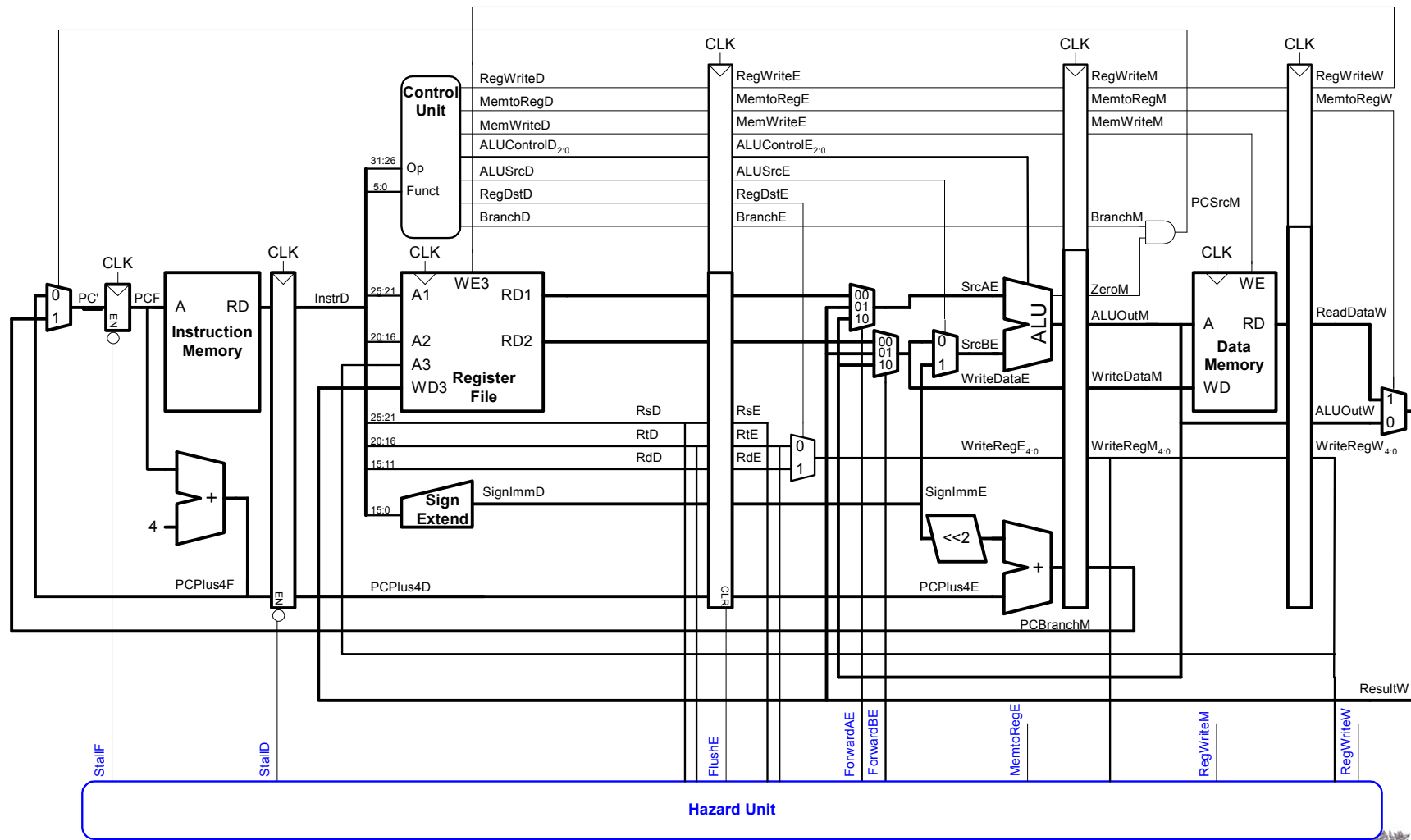
$((rsD == rtE) \text{ OR } (rtD == rtE)) \text{ AND } MemtoRegE$

*StallF* = *StallD* = *FlushE* = *lwstall*

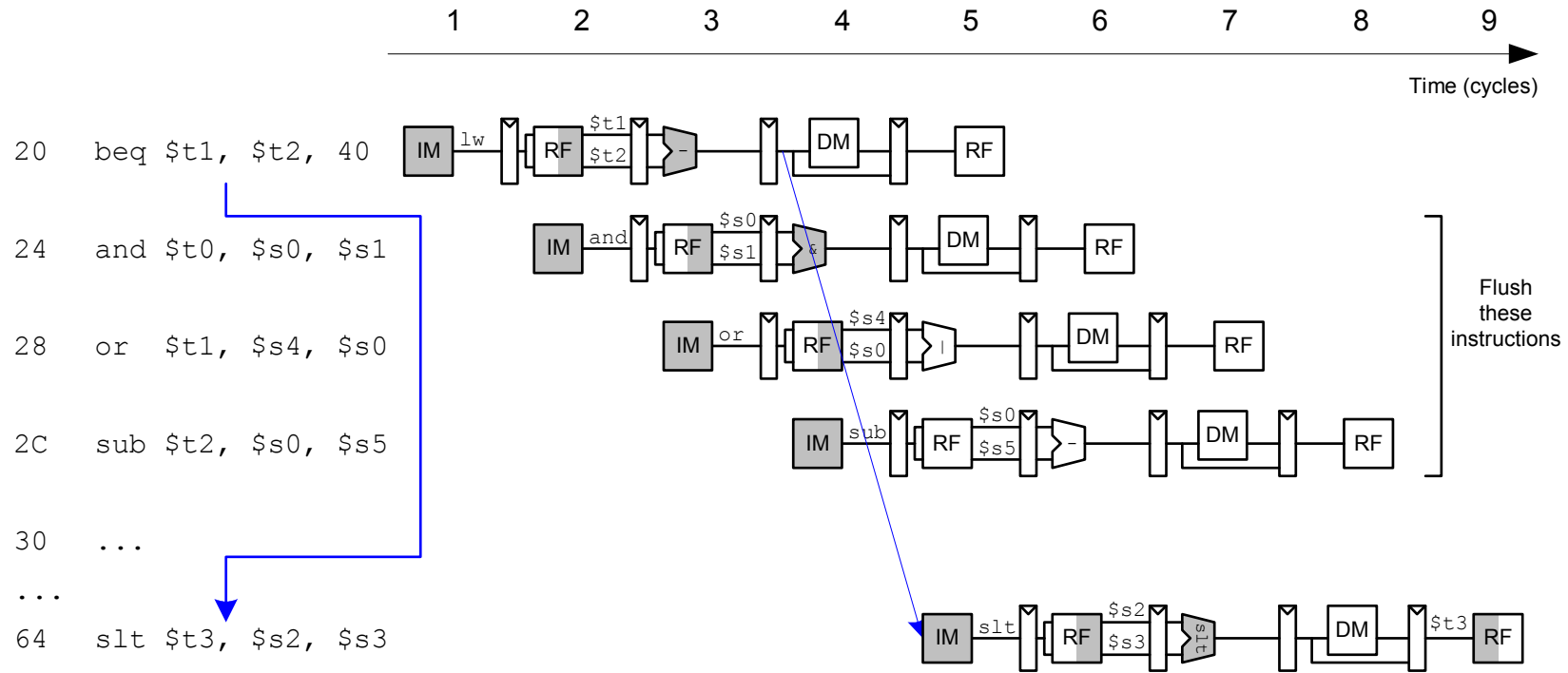
# Control Hazards

- **beq:**
  - branch not determined until 4<sup>th</sup> stage of pipeline
  - Instructions after branch fetched before branch occurs
  - These instructions must be flushed if branch happens
- **Branch misprediction penalty**
  - number of instruction flushed when branch is taken
  - May be reduced by determining branch earlier

# Control Hazards: Original Pipeline

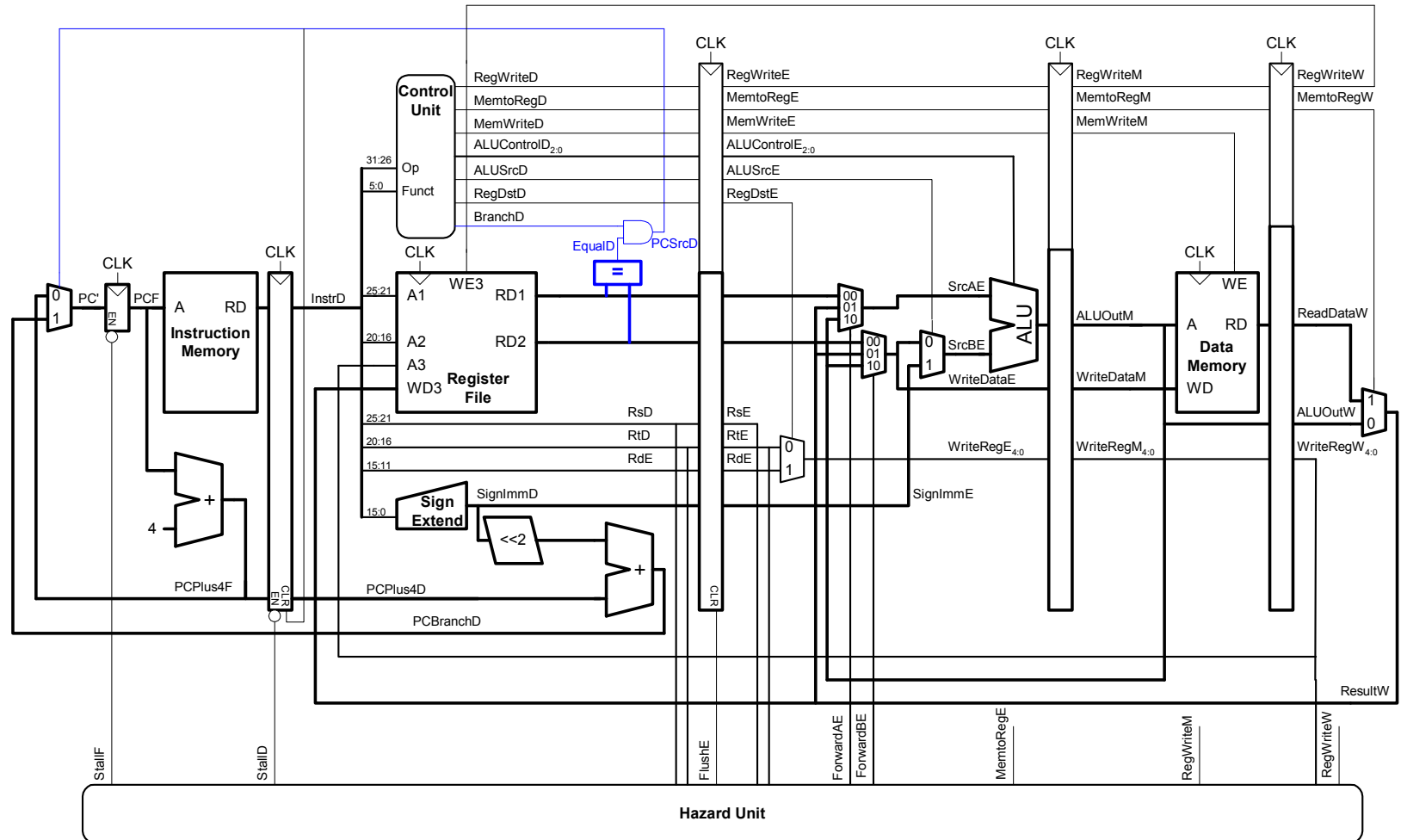


# Control Hazards



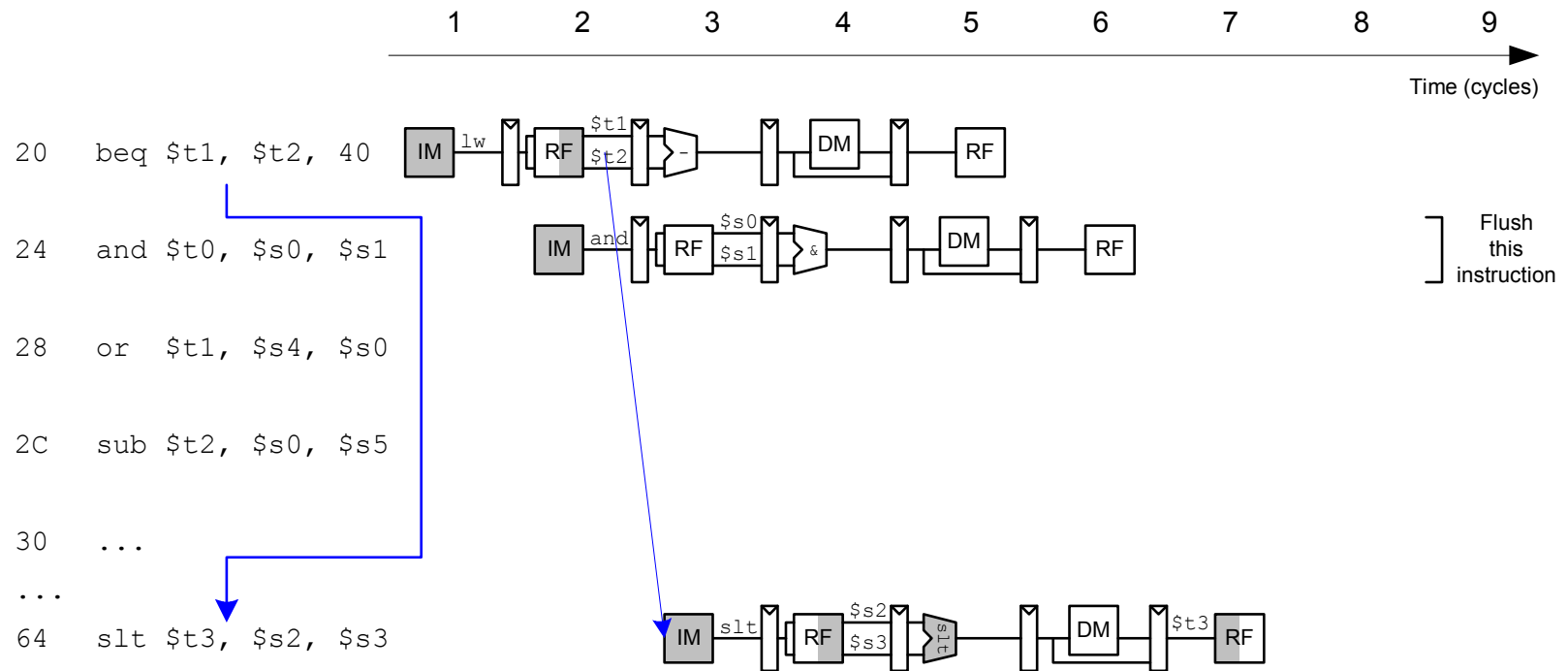


# Early Branch Resolution

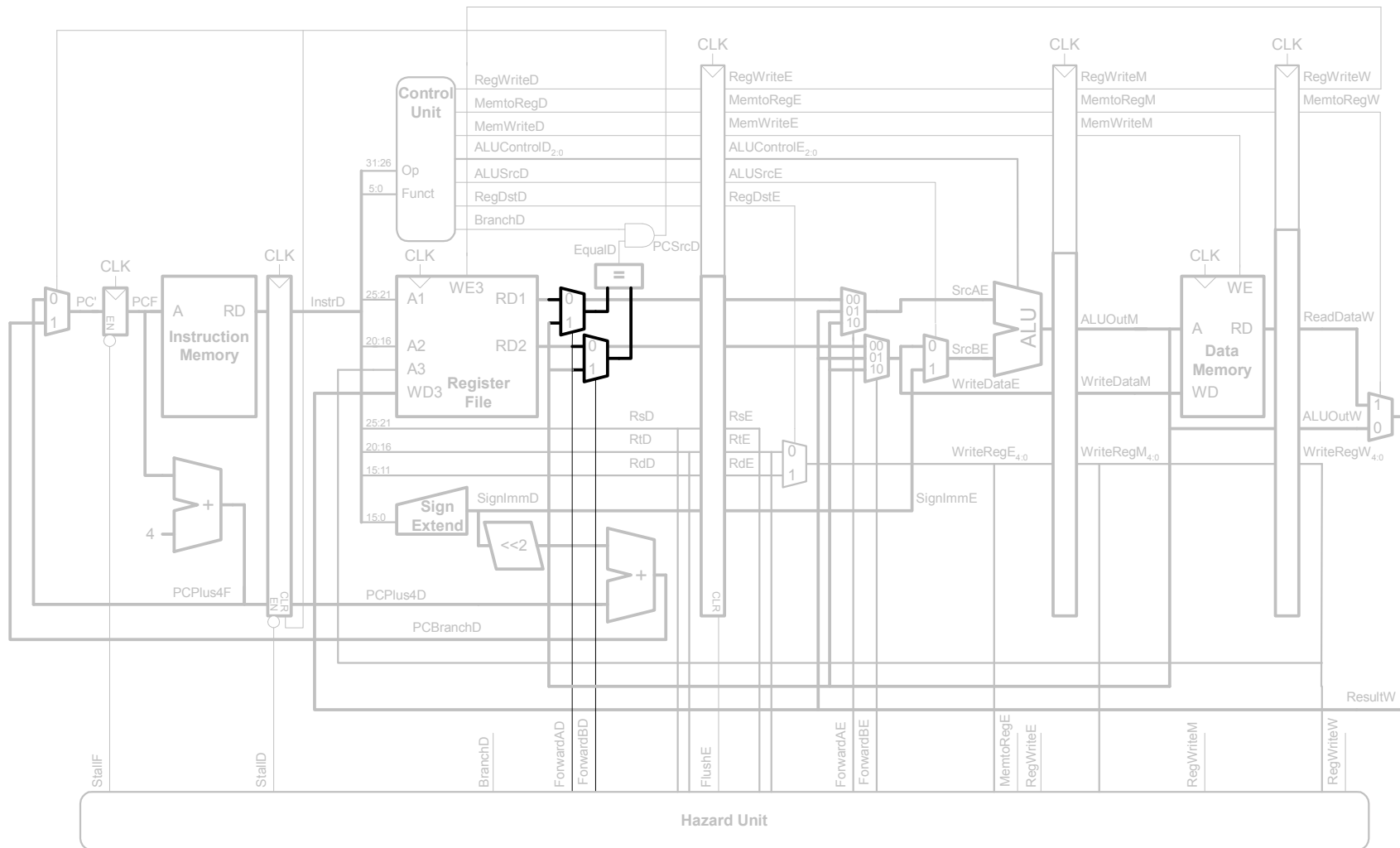


Introduced another data hazard in Decode stage

# Early Branch Resolution



# Handling Data & Control Hazards



# Control Forwarding & Stalling Logic

- **Forwarding logic:**

**ForwardAD** = (*rsD* !=0) AND (*rsD* == *WriteRegM*) AND *RegWriteM*

**ForwardBD** = (*rtD* !=0) AND (*rtD* == *WriteRegM*) AND *RegWriteM*

- **Stalling logic:**

**branchstall** = *BranchD* AND

[*RegWriteE* AND ((*WriteRegE* == *rsD*) OR (*WriteRegE* == *rtD*))

OR

[*MemtoRegM* AND ((*WriteRegM* == *rsD*) OR (*WriteRegM* == *rtD*))]

*StallF* = *StallD* = *FlushE* = *lwstall* OR *branchstall*

# Branch Prediction

- Guess whether branch will be taken
  - Backward branches are usually taken (loops)
  - Consider history to improve guess
- Good prediction reduces fraction of branches requiring a flush

# Pipelined Performance Example

- SPECINT2000 benchmark:
  - 25% loads
  - 10% stores
  - 11% branches
  - 2% jumps
  - 52% R-type
- Suppose:
  - 40% of loads used by next instruction
  - 25% of branches mispredicted
  - All jumps flush next instruction
- **What is the average CPI?**

# Pipelined Performance Example

- SPECINT2000 benchmark:
  - 25% loads
  - 10% stores
  - 11% branches
  - 2% jumps
  - 52% R-type
- Suppose:
  - 40% of loads used by next instruction
  - 25% of branches mispredicted
  - All jumps flush next instruction
- **What is the average CPI?**
  - Load/Branch CPI = 1 when no stalling, 2 when stalling
  - $CPI_{lw} = 1(0.6) + 2(0.4) = 1.4$
  - $CPI_{beq} = 1(0.75) + 2(0.25) = 1.25$

**Average CPI** =  $(0.25)(1.4) + (0.1)(1) + (0.11)(1.25) + (0.02)(2) + (0.52)(1)$

**= 1.15**

# Pipelined Performance

- Pipelined processor critical path:

$$T_c = \max \{$$

$$t_{pcq} + t_{mem} + t_{setup}$$

$$2(t_{RFread} + t_{mux} + t_{eq} + t_{AND} + t_{mux} + t_{setup})$$

$$t_{pcq} + t_{mux} + t_{mux} + t_{ALU} + t_{setup}$$

$$t_{pcq} + t_{memwrite} + t_{setup}$$

$$2(t_{pcq} + t_{mux} + t_{RFwrite}) \}$$



# Pipelined Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	$t_{pcq\_PC}$	30
Register setup	$t_{setup}$	20
Multiplexer	$t_{mux}$	25
ALU	$t_{ALU}$	200
Memory read	$t_{mem}$	250
Register file read	$t_{RFread}$	150
Register file setup	$t_{RFsetup}$	20
Equality comparator	$t_{eq}$	40
AND gate	$t_{AND}$	15
Memory write	$t_{memwrite}$	220
Register file write	$t_{RFwrite}$	100

$$\begin{aligned}
 T_c &= 2(t_{RFread} + t_{mux} + t_{eq} + t_{AND} + t_{mux} + t_{setup}) \\
 &= 2[150 + 25 + 40 + 15 + 25 + 20] \text{ ps} = \mathbf{550 \text{ ps}}
 \end{aligned}$$

# Pipelined Performance Example

Program with 100 billion instructions

$$\begin{aligned}\text{Execution Time} &= (\# \text{ instructions}) \times \text{CPI} \times T_c \\ &= (100 \times 10^9)(1.15)(550 \times 10^{-12}) \\ &= \mathbf{63 \text{ seconds}}\end{aligned}$$

# Processor Performance Comparison

Processor	Execution Time (seconds)	Speedup (single-cycle as baseline)
<b>Single-cycle</b>	92.5	1
<b>Multicycle</b>	133	0.70
<b>Pipelined</b>	63	1.47

# Review: Exceptions

- Unscheduled function call to *exception handler*
- Caused by:
  - Hardware, also called an *interrupt*, e.g. keyboard
  - Software, also called *traps*, e.g. undefined instruction
- When exception occurs, the processor:
  - Records cause of exception (Cause register)
  - Jumps to exception handler (0x80000180)
  - Returns to program (EPC register)

# Example Exception

sequential circuits.]

Can we design a spiff

Figure 2.11 shows a inputs, A and B, and on box indicates that it is this case, the function is

KeyAccess



The network KeyServer, which is required by KeyServer controlled programs, cannot grant you permission to run this program. If you think you have received this message in error, please contact your KeyServer Administrator.

Visio.exe - Application Error



The exception unknown software exception (0xc06d007e) occurred in the application at location 0x7c81eb33.

OK

words, we say the output Y is a function of the two inputs A and B where the function performed is A OR B.]

The *implementation* of the combinational circuit is independent of its functionality. Figure 2.1 and Figure 2.2 show two possible implementa-

# Exception Registers

- Not part of register file
  - Cause
    - Records cause of exception
    - Coprocessor 0 register 13
  - EPC (Exception PC)
    - Records PC where exception occurred
    - Coprocessor 0 register 14
- Move from Coprocessor 0
  - `mfc0 $t0, Cause`
  - Moves contents of Cause into `$t0`

**mfc0**

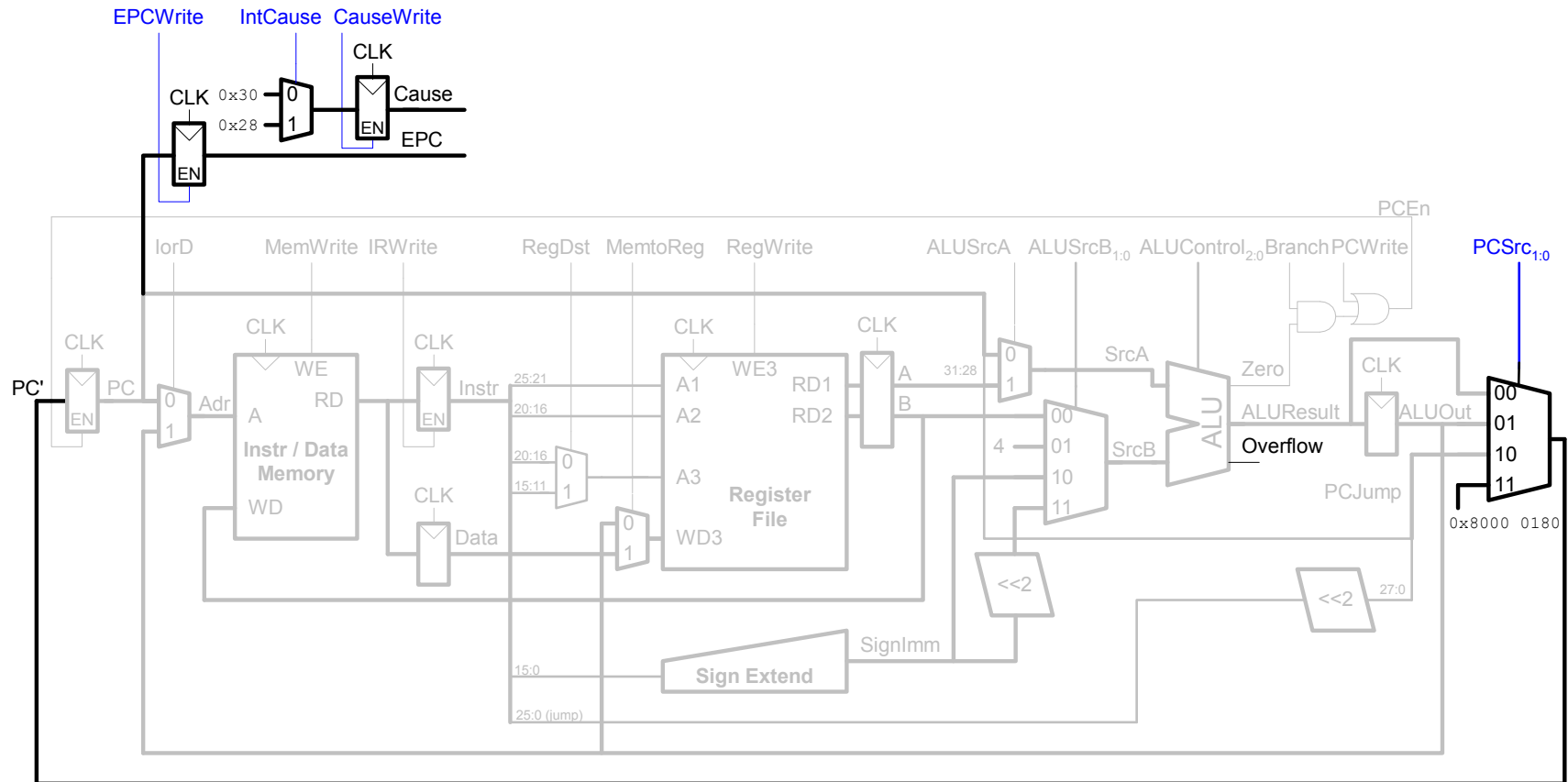
010000	00000	\$t0 (8)	Cause (13)	000000000000
31:26	25:21	20:16	15:11	10:0

# Exception Causes

Exception	Cause
Hardware Interrupt	0x00000000
System Call	0x00000020
Breakpoint / Divide by 0	0x00000024
<b>Undefined Instruction</b>	0x00000028
<b>Arithmetic Overflow</b>	0x00000030

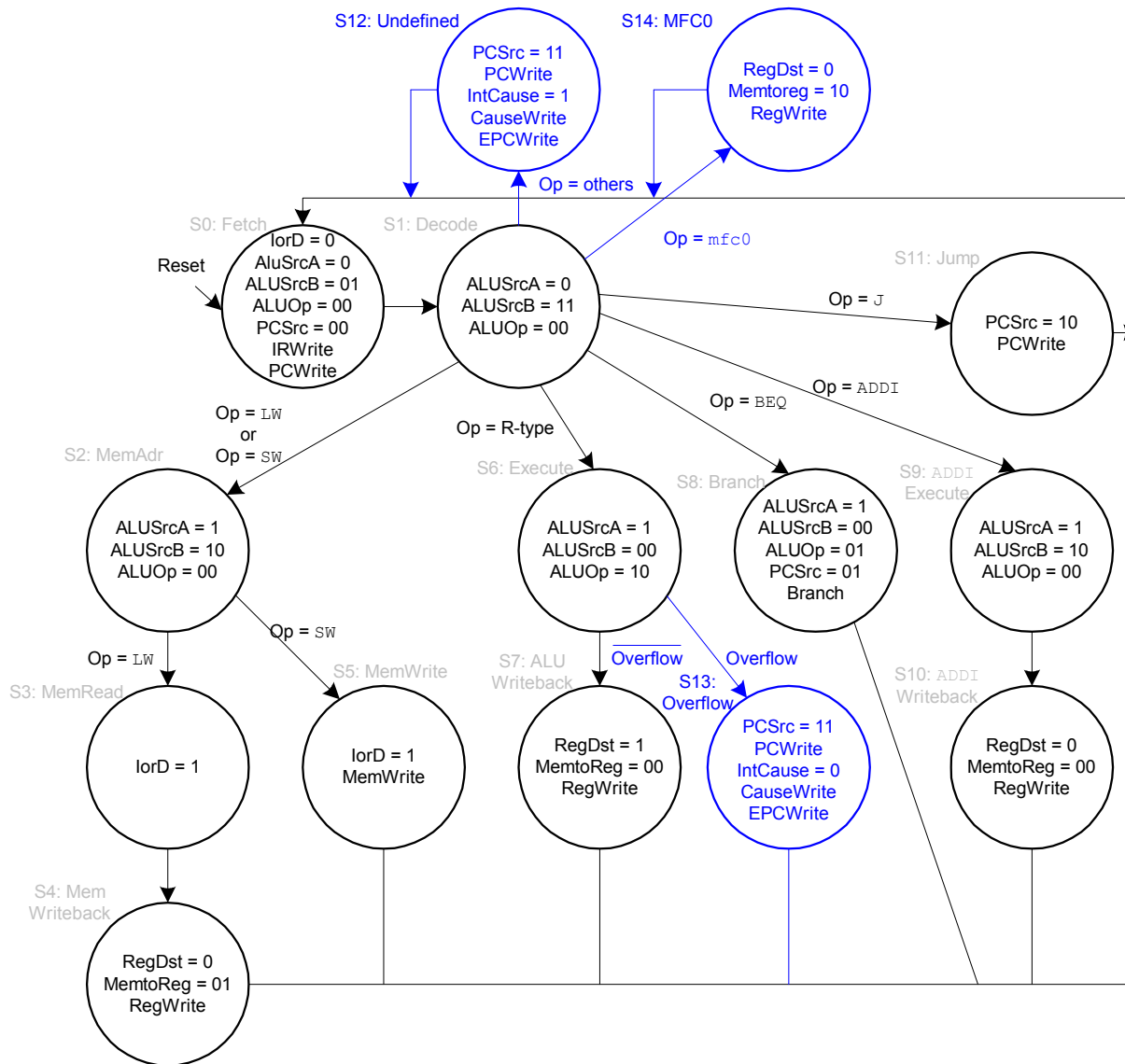
**Extend multicycle MIPS processor to handle last two types of exceptions**

# Exception Hardware: EPC & Cause

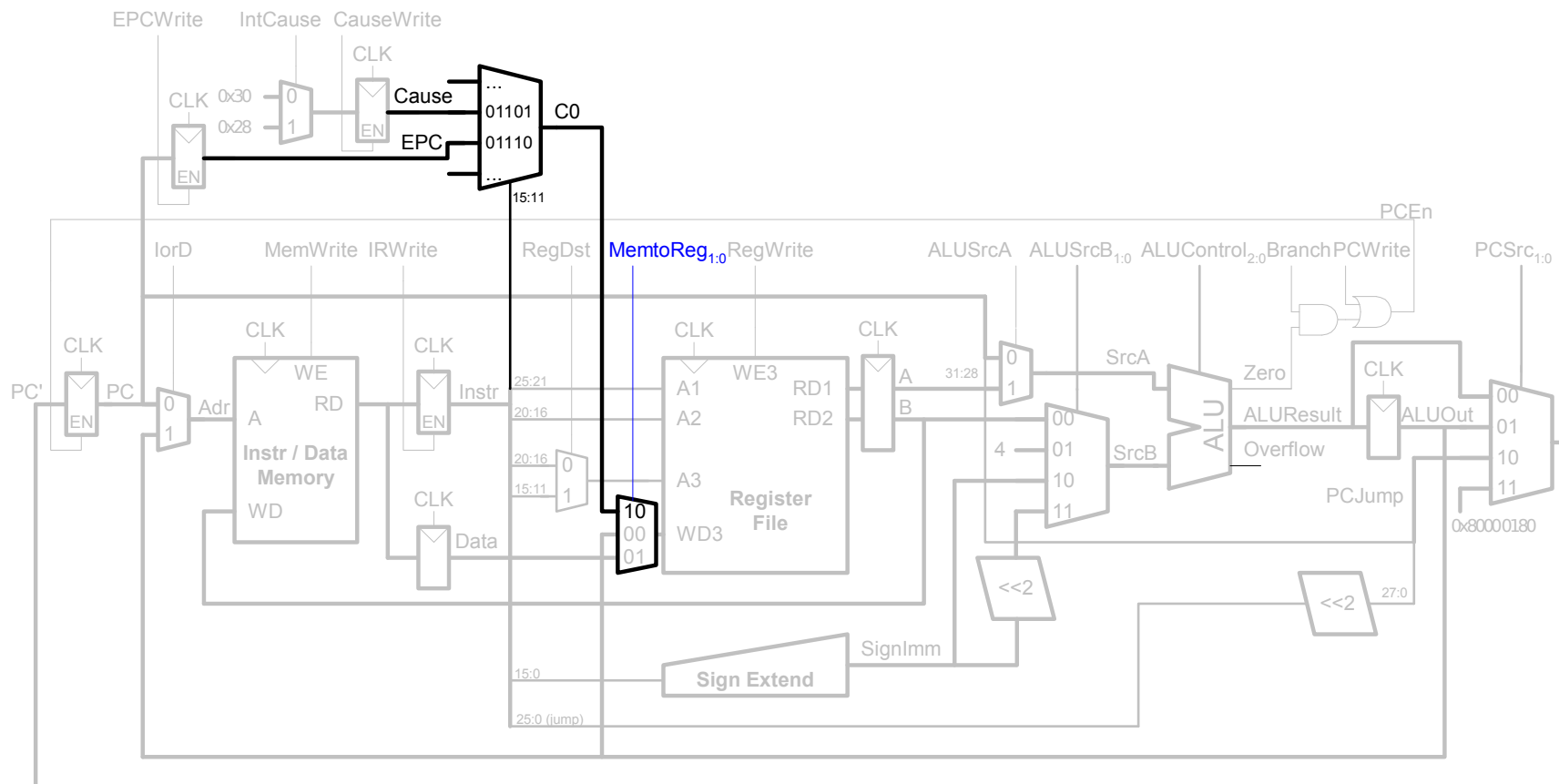




# Control FSM with Exceptions



# Exception Hardware: mfc0



# Advanced Microarchitecture

- Deep Pipelining
- Branch Prediction
- Superscalar Processors
- Out of Order Processors
- Register Renaming
- SIMD
- Multithreading
- Multiprocessors

# Deep Pipelining

- 10-20 stages typical
- Number of stages limited by:
  - Pipeline hazards
  - Sequencing overhead
  - Power
  - Cost

# Branch Prediction

- Ideal pipelined processor:  $CPI = 1$
- Branch misprediction increases CPI
- **Static branch prediction:**
  - Check direction of branch (forward or backward)
  - If backward, predict taken
  - Else, predict not taken
- **Dynamic branch prediction:**
  - Keep history of last (several hundred) branches in *branch target buffer*, record:
    - Branch destination
    - Whether branch was taken



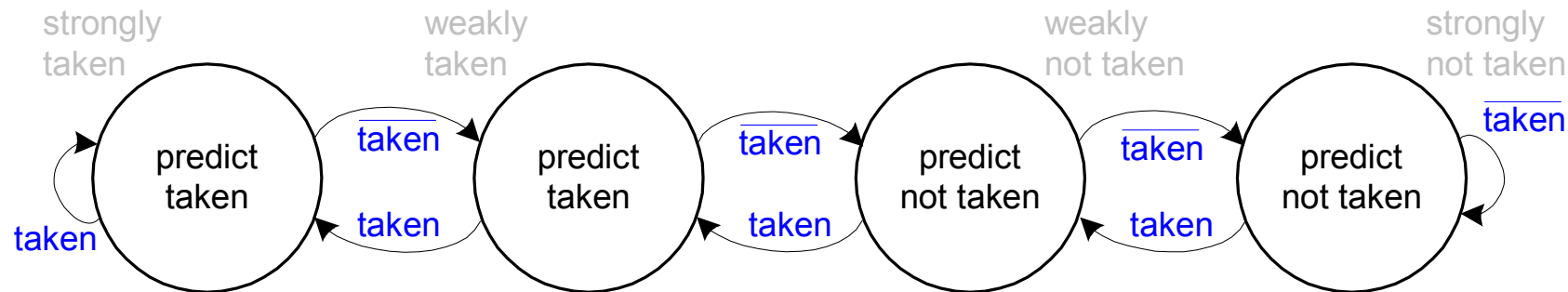
# Branch Prediction Example

```
add    $s1, $0, $0        # sum = 0
add    $s0, $0, $0        # i   = 0
addi   $t0, $0, 10        # $t0 = 10
for:
    beq  $s0, $t0, done    # if i == 10, branch
    add  $s1, $s1, $s0     # sum = sum + i
    addi $s0, $s0, 1       # increment i
    j    for
done:
```

# 1-Bit Branch Predictor

- Remembers whether branch was taken the last time and does the same thing
- Mispredicts first and last branch of loop

# 2-Bit Branch Predictor

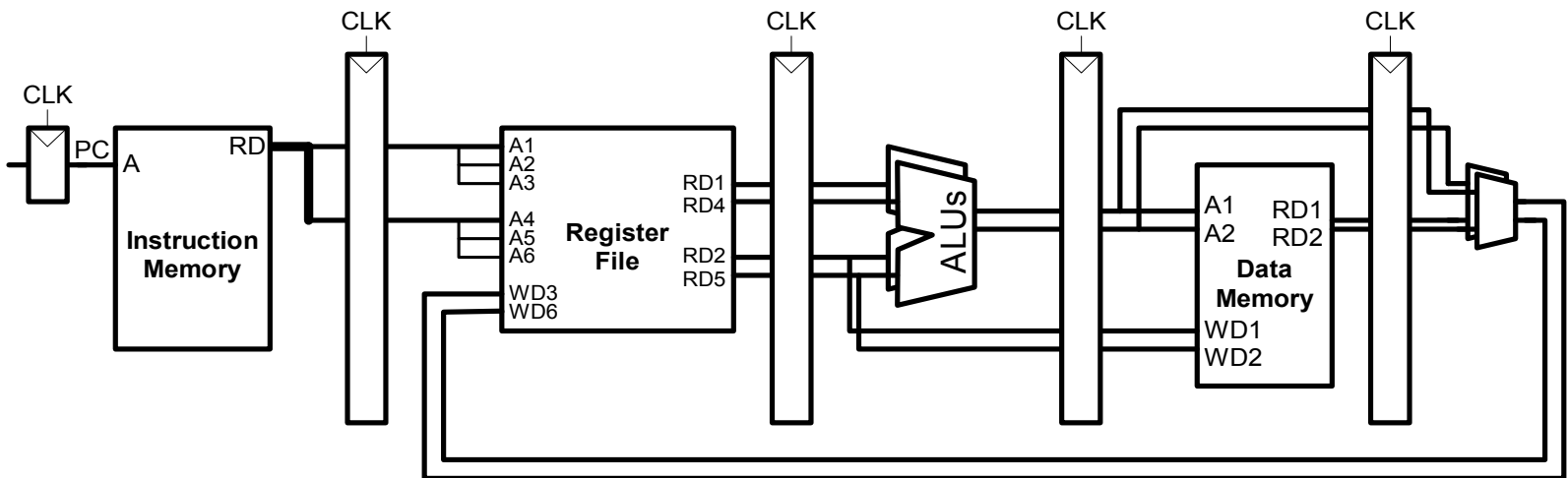


**Only mispredicts last branch of loop**



# Superscalar

- Multiple copies of datapath execute multiple instructions at once
- Dependencies make it tricky to issue multiple instructions at once

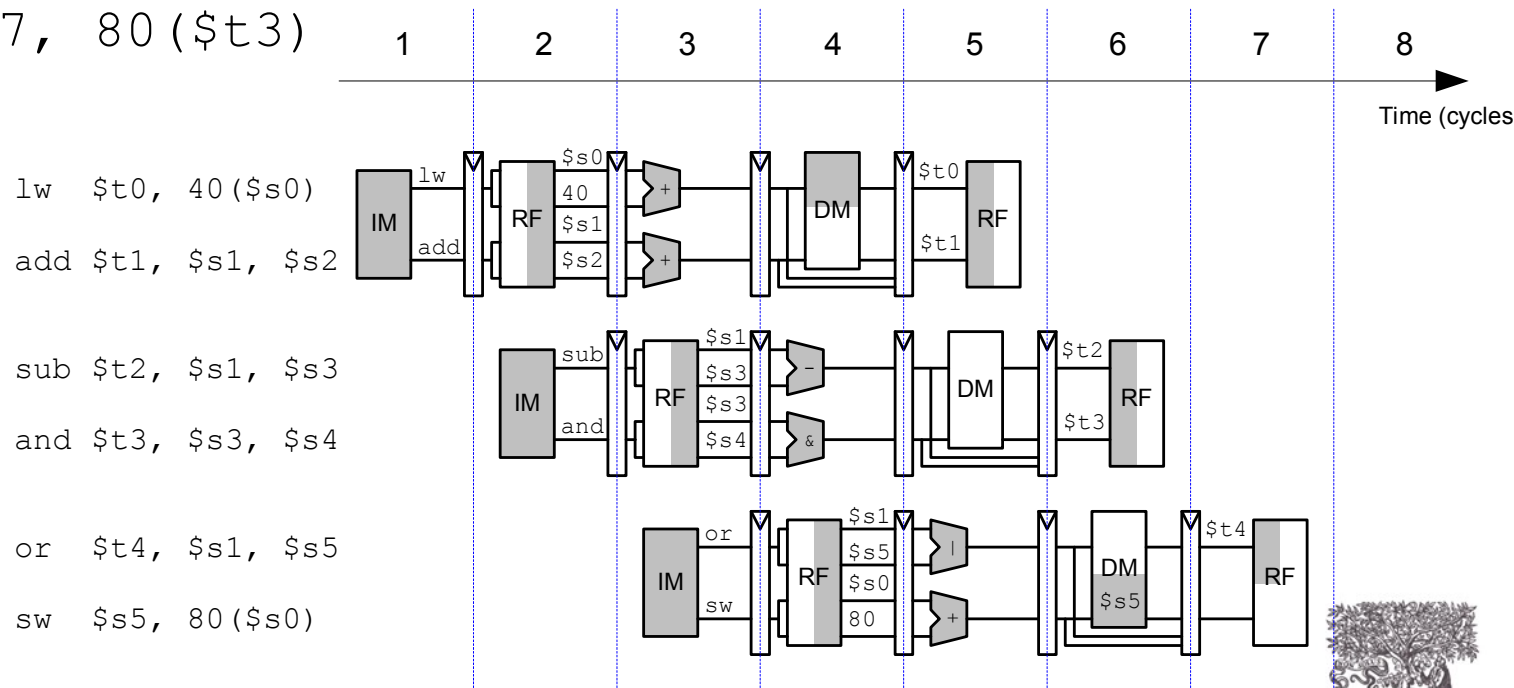


# Superscalar Example

```
lw    $t0, 40($s0)
add   $t1, $t0, $s1
sub   $t0, $s2, $s3
and   $t2, $s4, $t0
or    $t3, $s5, $s6
sw    $s7, 80($t3)
```

Ideal IPC: 2

Actual IPC: 2

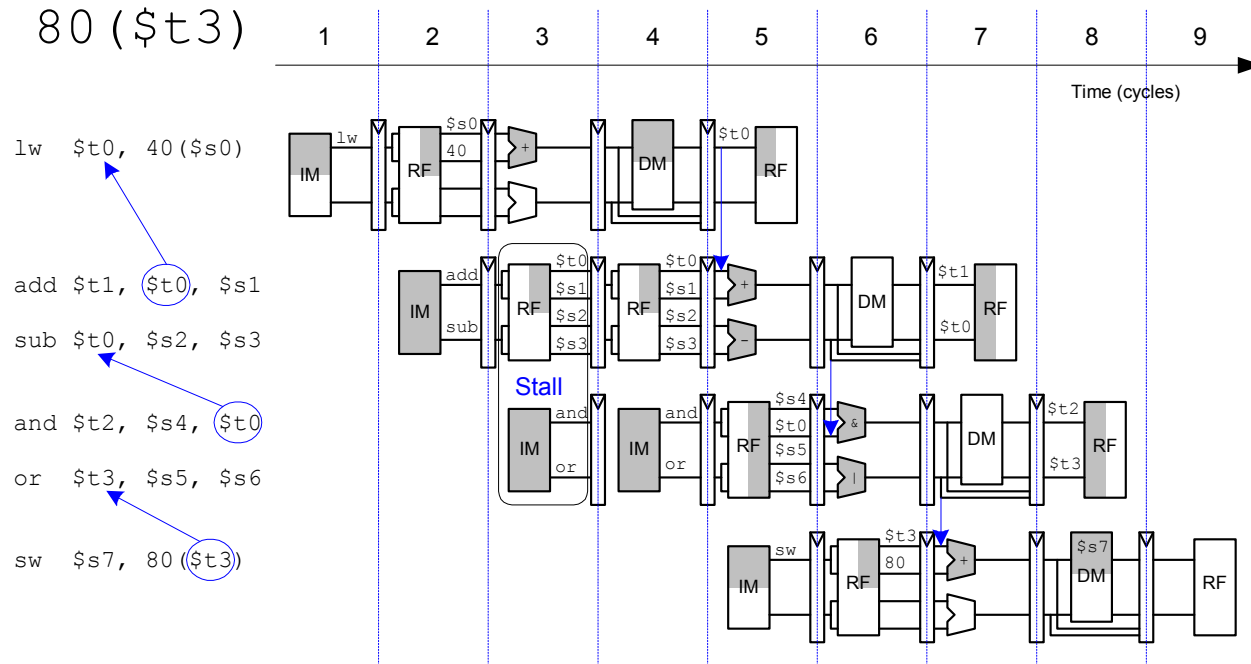


# Superscalar with Dependencies

```
lw    $t0, 40($s0)
add   $t1, $t0, $s1
sub   $t0, $s2, $s3
and   $t2, $s4, $t0
or    $t3, $s5, $s6
sw    $s7, 80($t3)
```

Ideal IPC: 2

Actual IPC:  $6/5 = 1.2$



# Out of Order Processor

- Looks ahead across multiple instructions
- Issues as many instructions as possible at once
- Issues instructions out of order (as long as no dependencies)
- **Dependencies:**
  - **RAW** (read after write): one instruction writes, later instruction reads a register
  - **WAR** (write after read): one instruction reads, later instruction writes a register
  - **WAW** (write after write): one instruction writes, later instruction writes a register



# Out of Order Processor

- **Instruction level parallelism (ILP):** number of instruction that can be issued simultaneously (average  $< 3$ )
- **Scoreboard:** table that keeps track of:
  - Instructions waiting to issue
  - Available functional units
  - Dependencies

# Out of Order Processor Example

```

lw    $t0, 40($s0)
add   $t1, $t0, $s1
sub   $t0, $s2, $s3
and   $t2, $s4, $t0
or    $t3, $s5, $s6
sw    $s7, 80($t3)

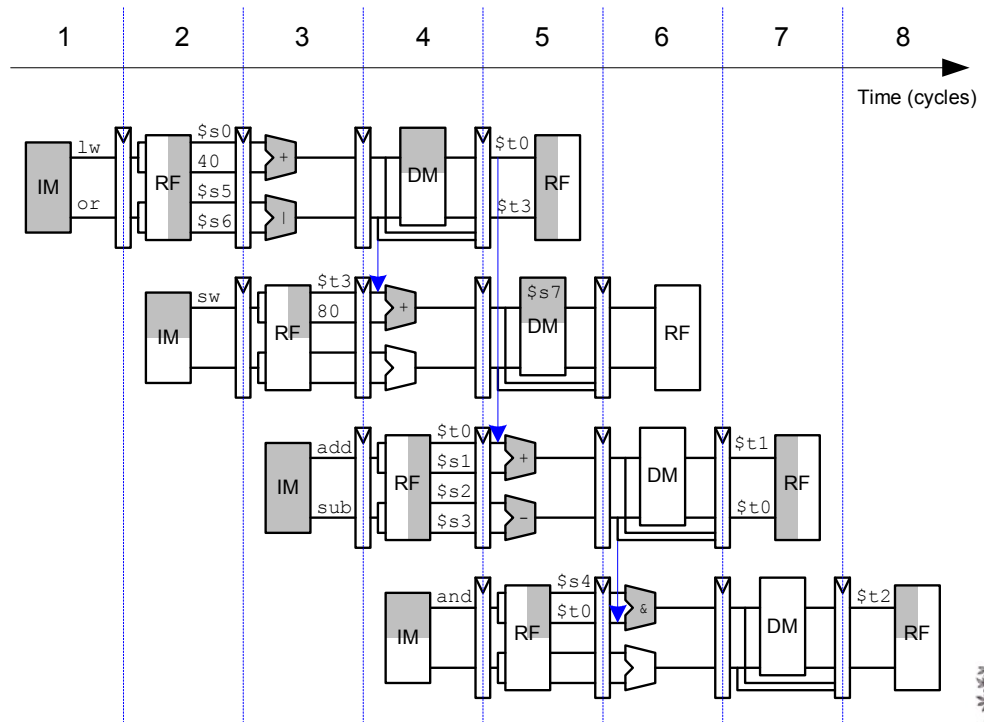
```

Ideal IPC: 2

Actual IPC:  $6/4 = 1.5$

lw \$t0, 40(\$s0)  
 or \$t3, \$s5, \$s6  
 sw \$s7, 80(\$t3)  
 add \$t1, \$t0, \$s1  
 sub \$t0, \$s2, \$s3  
 and \$t2, \$s4, \$t0

RAW (between lw and or)  
 RAW (between lw and sw)  
 two cycle latency between load and use of \$t0  
 WAR (between add and sub)  
 RAW (between sub and and)

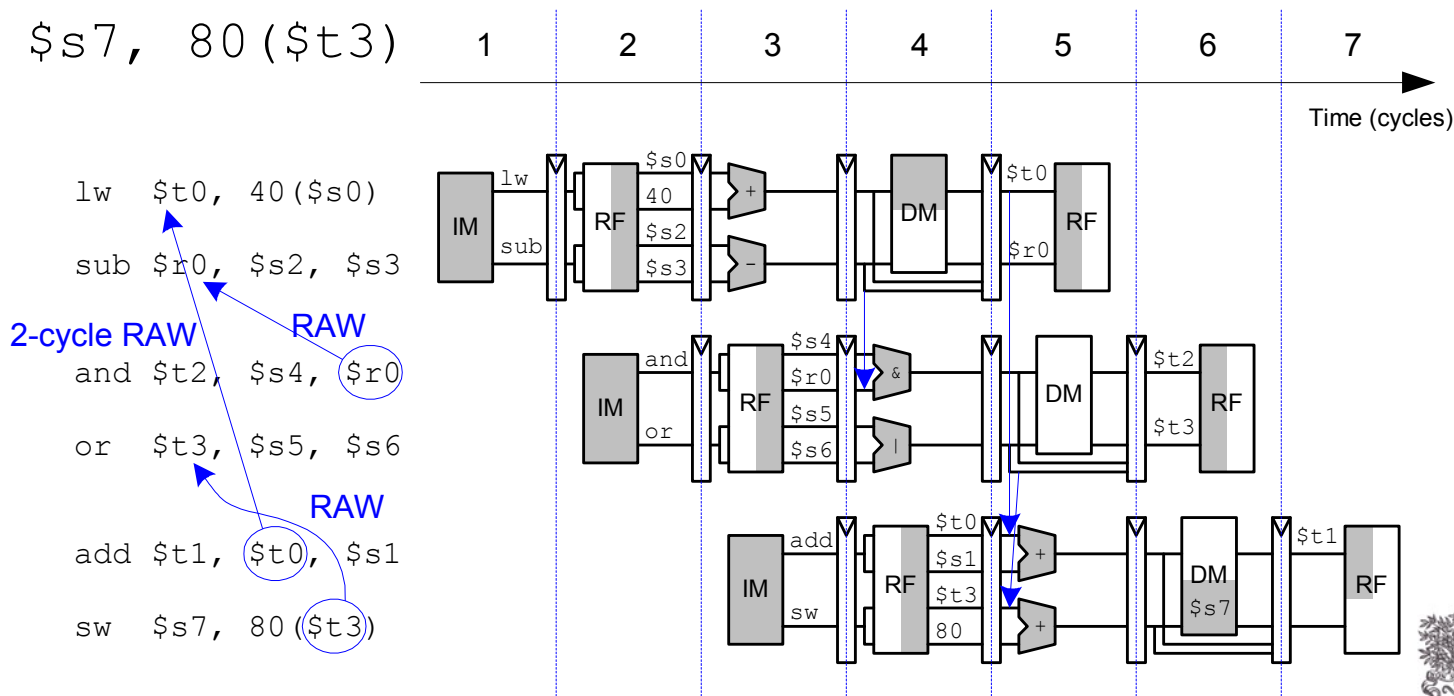


# Register Renaming

```
lw    $t0, 40($s0)
add   $t1, $t0, $s1
sub    $t0, $s2, $s3
and    $t2, $s4, $t0
or     $t3, $s5, $s6
sw     $s7, 80($t3)
```

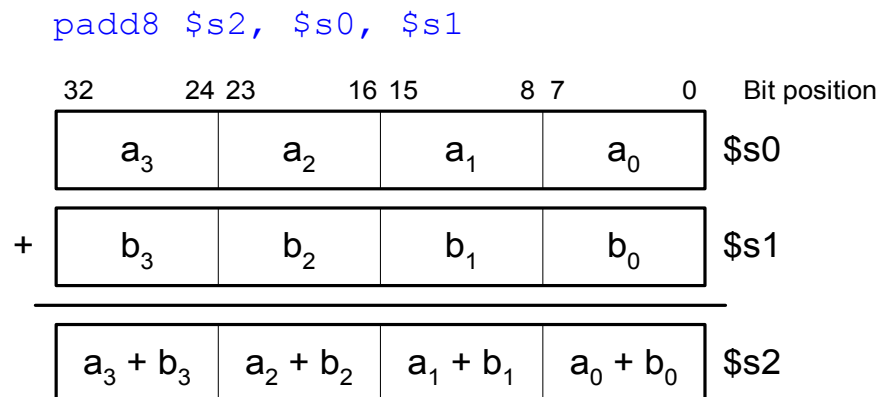
Ideal IPC: 2

Actual IPC:  $6/3 = 2$



## SIMD

- Single Instruction Multiple Data (SIMD)
  - Single instruction acts on multiple pieces of data at once
  - Common application: graphics
  - Perform short arithmetic operations (also called *packed arithmetic*)
- For example, add four 8-bit elements





# Advanced Architecture Techniques

- **Multithreading**
  - Wordprocessor: thread for typing, spell checking, printing
- **Multiprocessors**
  - Multiple processors (cores) on a single chip

# Threading: Definitions

- **Process:** program running on a computer
  - Multiple processes can run at once: e.g., surfing Web, playing music, writing a paper
- **Thread:** part of a program
  - Each process has multiple threads: e.g., a word processor may have threads for typing, spell checking, printing

# Threads in Conventional Processor

- One thread runs at once
- When one thread stalls (for example, waiting for memory):
  - Architectural state of that thread stored
  - Architectural state of waiting thread loaded into processor and it runs
  - Called **context switching**
- Appears to user like all threads running simultaneously

# Multithreading

- Multiple copies of architectural state
- Multiple threads **active** at once:
  - When one thread stalls, another runs immediately
  - If one thread can't keep all execution units busy, another thread can use them
- Does not increase instruction-level parallelism (ILP) of single thread, but increases throughput

**Intel calls this “hyperthreading”**

# Multiprocessors

- Multiple processors (cores) with a method of communication between them
- Types:
  - **Homogeneous:** multiple cores with shared memory
  - **Heterogeneous:** separate cores for different tasks (for example, DSP and CPU in cell phone)
  - **Clusters:** each core has own memory system

# Other Resources

- Patterson & Hennessy's: *Computer Architecture: A Quantitative Approach*
- Conferences:
  - [www.cs.wisc.edu/~arch/www/](http://www.cs.wisc.edu/~arch/www/)
  - ISCA (International Symposium on Computer Architecture)
  - HPCA (International Symposium on High Performance Computer Architecture)